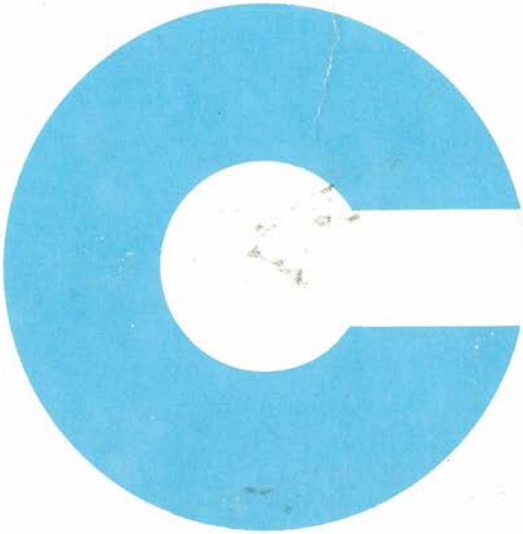
LINGUAGGIO

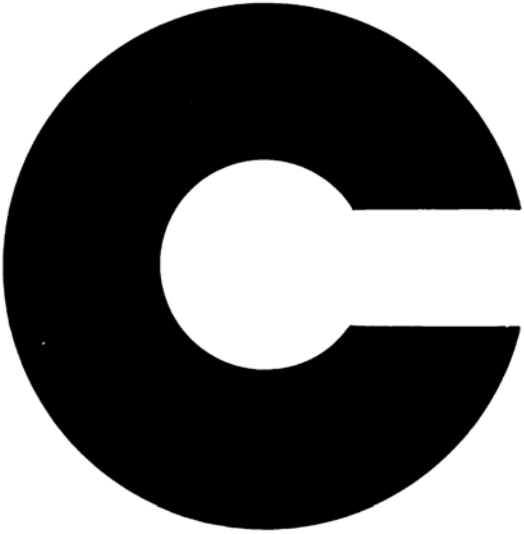


Brian W. Kernighan
Dennis M. Ritchie

EDIZIONE ITALIANA

GRUPPO
EDITORIALE
JACKSON

LINGUAGGIO



Brian W. Kernighan
Dennis M. Ritchie

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

© Copyright per l'edizione originale: PRENTICE - HALL inc. 1978
© Copyright per l'edizione originale: Gruppo Editoriale Jackson - Marzo 1985
SUPERVISIONE TECNICA: Daria Gianni
GRAFICA E IMPAGINAZIONE: Francesca Di Fiore
COPERTINA: Silvana Corbelli
FOTOCOMPOSIZIONE: System Graphic S.r.l. - Cologno Monzese (MI)
STAMPA: Grafika '78 - Via Trieste, 20 - Pioltello (MI)

Tutti i diritti sono riservati. Stampato in Italia. Nessuna parte di questo libro può essere riprodotta, memorizzata in sistemi di archivio, o trasmessa in qualsiasi forma o mezzo, elettronico, meccanico, fotocopia, registrazione o altri senza la preventiva autorizzazione scritta dell'editore.

SOMMARIO

INTRODUZIONE ALL'EDIZIONE ITALIANA IX
PREFAZIONE XI

Capitolo 0 - INTRODUZIONE 1

Capitolo 1 - UN'INTRODUZIONE DIDATTICA 5
1.1 Per cominciare ... 5
1.2 Variabili e Aritmetica ... 8
1.3 L'istruzione for ... 11
1.4 Costanti simboliche ... 12
1.5 Una raccolta di programmi utili ... 13
1.6 Arrays ... 20
1.7 Funzioni ... 22
1.8 Argomenti - Chiamata per Valore ... 23
1.9 Array di caratteri ... 24
1.10 Visibilità; Variabili Esterne ... 27
1.11 Sommario ... 30

Capitolo 2 - TIPI, OPERATORI ED ESPRESSIONI 31
2.1 Nomi di variabili ... 31
2.2 Tipi e dimensioni dei dati ... 31
2.3 Le costanti ... 32
2.4 Le dichiarazioni ... 34
2.5 Operatori Aritmetici ... 35
2.6 Operatori Relazionali e Logici ... 36
2.7 Conversioni di Tipo ... 37
2.8 Operatori di Incremento e Decremento ... 40
2.9 Operatori Logici orientati al Bit ... 42
2.10 Operatori ed Espressioni di Assegnamento ... 44
2.11 Espressioni Condizionali ... 45
2.12 Priorità e Ordine di valutazione ... 46

Capitolo 3 - STRUTTURE DI CONTROLLO 49
3.1 Istruzioni e Blocchi ... 49
3.2 If - Else ... 49
3.3 Else - If ... 51
3.4 Switch ... 52
3.5 Loops - While e For ... 53
3.6 Loops - Do-while ... 56
3.7 Break ... 57
3.8 Continue ... 58
3.9 Goto e Label ... 59

Capitolo 4 - FUNZIONI E STRUTTURA DEI PROGRAMMI 61
4.1 Fondamenti ... 61
4.2 Funzioni che non ritornano interi ... 64
4.3 Ulteriori precisazioni sugli argomenti di funzione ... 66
4.4 Variabili Esterne ... 67
4.5 Regole di Visibilità ... 71
4.6 Variabili static ... 74
4.7 Variabili Register ... 75
4.8 Struttura a Blocchi ... 76
4.9 Inizializzazione ... 77
4.10 Ricorsione ... 79
4.11 Il Preprocessor C ... 80

Capitolo 5 - PUNTATORI ED ARRAY 83
5.1 Puntatori e Indirizzi ... 83
5.2 Puntatori e Argomenti di Funzioni ... 85
5.3 Puntatori ed Array ... 87
5.4 Aritmetica sugli Indirizzi ... 90
5.5 Puntatori a caratteri e Funzioni ... 93
5.6 I Puntatori non sono Interi ... 96
5.7 Array Multi-dimensionali ... 97
5.8 Array di Puntatori; Puntatori a Puntatori ... 98
5.9 Inizializzazione degli array di Puntatori ... 101
5.10 Confronto tra Puntatori e Array Multi-dimensionali ... 102
5.11 Argomenti sulla linea di comando ... 103
5.12 Puntatori a Funzioni ... 107

Capitolo 6 - STRUTTURE 111
6.1 Fondamenti ... 111
6.2 Strutture e funzioni ... 113
6.3 Arrays di strutture ... 115
6.4 Puntatori alle strutture ... 119
6.5 Strutture ricorsive ... 121
6.6 Ricerca tabellare ... 125
6.7 Campi ... 127
6.8 Unioni ... 129
6.9 Typedef ... 130

Capitolo 7 - INPUT ED OUTPUT 133
7.1 Accesso alla Libreria Standard ... 133
7.2 Standard Input ed Output - Getchar e Putchar ... 134
7.3 Output Formattato - Printf ... 135
7.4 Input Formattato - Scanf ... 137
7.5 Conversione di Formati di Memoria ... 139
7.6 Accesso a File ... 140
7.7 Gestione degli Errori - Stderr e Exit ... 143
7.8 Input e Output di Linea ... 144
7.9 Altre Funzioni ... 145

Capitolo 8 - L'INTERFACCIA CON IL SISTEMA UNIX 147
8.1 Descrittori di file ... 147
8.2 1/0 a basso livello - Read e Write ... 148
8.3 Open, Creat, Close, Unlink ... 150
8.4 Accesso Random - Seek e Lseek ... 151
8.5 Esempio - Un'Implementazione di Fopen e Getc ... 152
8.6 Esempio - Lista di Directory ... 156
8.7 Esempio - Un allocatore di memoria ... 159

Appendice A - MANUALE DI RIFERIMENTO DEL C 165
1. Introduzione ... 165
2. Convenzioni Lessicali ... 165
2.1 Commenti ... 165
2.2 Identificatori (Nomi) ... 165
2.3 Parole Chiave ... 166
2.4 Costanti ... 166
2.4.1 Costanti Intere ... 166
2.4.2 Costanti long esplicite ... 166
2.4.3 Costanti Carattere ... 166
2.4.4 Costanti Floating ... 167
2.5 Stringhe ... 167
2.6 Caratteristiche Hardware ... 167
3. Notazione Sintattica ... 167
4. Cosa c'è in un nome? ... 168
5. Oggetti e lvalue ... 169
6. Conversioni ... 169
6.1 Caratteri ed interi ... 169
6.2 Float e double ... 169
6.3 Floating e integral ... 169
6.4 Puntatori e interi ... 170
6.5 Unsigned ... 170
6.6 Conversioni aritmetiche ... 170
7. Espressioni ... 170
7.1 Espressioni primarie ... 171
7.2 Operatori unari ... 173
7.3 Operatori moltiplicativi ... 174
7.4 Operatori additivi ... 174
7.5 Operatori di shift ... 175
7.6 Operatori relazionali ... 175
7.7 Operatori di eguaglianza ... 175
7.8 Operatore AND sui bit ... 176

7.9 Operatore OR esclusivo sui bit 176
7.10 Operatore OR inclusivo sui bit 176
7.11 Operatore AND logico 176
7.12 Operatore OR logico 177
7.13 Operatore condizionale 177
7.14 Operatori di assegnamento 177
7.15 Operatore virgola 178
8. Dichiarazioni 178
8.1 Specifiche di classe di memoria 179
8.2 Specifiche di tipo 179
8.3 Dichiaratori 180
8.4 Significato dei dichiaratori 180
8.5 Dichiarazione di strutture e unioni 182
8.6 Inizializzazione 184
8.7 Nomi di tipo 186
8.8 Typedef 187
9. Istruzioni 187
9.1 Istruzione espressione 187
9.2 Istruzioni composte o blocco 188
9.3 Istruzione condizionale 188
9.4 Istruzione while 188
9.5 Istruzione do 188
9.6 Istruzione for 189
9.7 Istruzione switch 189
9.8 Istruzione break 190
9.9 Istruzione continue 190
9.10 Istruzione return 190
9.11 Istruzione goto 190
9.12 Istruzione con label 191
9.13 Istruzione nulla 191
10. Definizioni esterne 191
10.1 Definizioni di funzioni esterne 191
10.2 Definizioni di dati esterni 192
11. Regole di visibilità 192
11.1 Visibilità lessicale 193
11.2 Visibilità delle esterne 193
12. Linee di controllo del compilatore 194
12.1 Rimpiazzamento di simboli 194
12.2 Inclusione di file 195
12.3 Compilazione condizionale 195
12.4 Controllo di linea 195
13. Dichiarazioni implicite 196
14. Analisi dei tipi 196
14.1 Strutture e unioni 196
14.2 Funzioni 196
14.3 Array, puntatori ed indicizzazione 197
14.4 Conversione esplicita di puntatori 197
15. Espressioni costanti 198

16. Considerazioni di portabilità 199
17. Anacronismi 200
18. Sommario Sintattico 200
18.1 Espressioni 200
18.2 Dichiarazioni 202
18.3 Istruzioni 203
18.4 Definizioni esterne 204
18.5 Preprocessore 204

INTRODUZIONE ALL'EDIZIONE ITALIANA

Il C si sta affermando in questi anni come un linguaggio di programmazione non più relegato ad una stretta cerchia di applicazioni: è stato infatti fino ad ora definito un "linguaggio di programmazione di sistema", idoneo allo sviluppo di sistemi operativi e di software di base. Il suo sviluppo corre in parallelo a quello del sistema operativo UNIX*, il cui nucleo centrale e la gran parte dei programmi di utilità sono scritti in C. UNIX ed i suoi derivati rappresentano un'ideale simbiosi tra linguaggio e sistema per la presenza di numerosi tools dedicati alla produzione di software C. Nell'ambito dei personal e micro, la massiccia introduzione del sistema operativo MS-DOS da parte dei costruttori sta dando una forte spinta ad una più diffusa conoscenza del linguaggio. UNIX e MS-DOS si stanno imponendo come standard rispettivamente nell'ambito dei supermicro e mini da una parte e personal e micro dall'altra, anche se oggi come oggi con l'introduzione degli ultimi microprocessori e memorie queste distinzioni di fascia hanno un peso sempre minore.

La caratteristica che più di ogni altra ha permesso lo sviluppo del C è la portabilità. Nel marasma di macchine, sistemi operativi e microprocessori, è generalmente arduo trovare un compilatore compatibile con un'altro. Il compilatore C si presenta invece omogeneo e compatto sotto qualsiasi ambiente permettendo il trasporto praticamente completo dei sorgenti. Pensare di produrre software per tutte le macchine invece che per una sola è certamente un fattore di veloce arricchimento per una biblioteca di applcazioni.

Il primo impatto per chi non conosce il linguaggio non è dei più immediati. La sintassi non è certo elegante e rigorosa come quella di altri linguaggi strutturati come Pascal: il C è estremamente sintetico e a prima vista può risultare poco leggibile, ma permette al programmatore esperto di mantenere "brevi" funzioni anche molto complesse. In definitiva, non è un linguaggio didattico, anche se il suo apprendimento è tutt'altro che ostico a chi sia in possesso di una certa cultura sui linguaggi strutturati, rappresentando un passo in avanti in termini di efficienza e di flessibilità. Il C offre una libertà di programmazione che viene generalmente apprezzata quando si è raggiunto un certo grado di esperienza poiché il compilatore è di conseguenza piuttosto avaro di segnalazioni di errore ed avvertenze.

Generalmente dove esiste il C esistono anche svariate librerie facilmente accessibili che vanno dalle applicazioni matematiche alla gestione di data-base, dalle funzioni per la gestione del video alle chiamate al sistema operativo, dalla comunicazione tra processi asincroni a funzioni di sort. La velocità di esecuzione è molto alta se confrontata con quella di altri linguaggi ad alto livello compilati.

Questo libro è il testo "ufficiale" del linguaggio, un prezioso punto di riferimento per tutti coloro che hanno a che fare col C. Il testo italiano è stato mantenuto il più possibile fedele all'originale, specialmente laddove era necessario rispettare il rigore di alcune definizioni. Nei numerosi programmi presenti, sono stati tradotti gli identificatori ed i nomi delle funzioni che non fanno parte delle librerie.

Maurizio Guazzone – Ivo Quartiroli

PREFAZIONE

Il C è un linguaggio di programmazione di applicazione generale che presenta come proprie caratteristiche la sinteticità delle espressioni, moderne strutture di controllo e di dati, e un esteso insieme di operatori. Il C non è un linguaggio di "altissimo livello", né un linguaggio "grosso" e non è specializzato per nessuna particolare applicazione. Ma l'assenza di restrizioni e la sua generalità fa sì che risulti per molti lavori più conveniente ed efficiente di linguaggi sulla carta più potenti.
Il C è stato originariamente progettato ed implementato da Dennis Ritchie per il sistema operativo UNIX (1). Il sistema operativo, il compilatore C ed essenzialmente tutti i programmi applicativi di UNIX (incluso tutto il software usato per la preparazione di questo libro) sono scritti in C. Sono inoltre disponibili compilatori per molte altre macchine, tra cui IBM System/370, Honeywell 6000 ed Interdata 8/32. Il C non è comunque legato a particolari macchine o sistemi, ed è semplice scrivere programmi che possono essere eseguiti senza alcuna modifica su qualsiasi macchina che lo supporta.
Lo scopo di questo libro è di aiutare il lettore ad imparare a programmare in C. Contiene un'introduzione didattica per mettere in grado i nuovi utenti di usarlo il più presto possibile, capitoli dedicati ad ogni argomento principale, ed un manuale di riferimento. La maggior parte della trattazione è basata sulla lettura, scrittura e riesame di esempi, programmi completi e non frammenti isolati, piuttosto che su un'arida enunciazione di regole. Tutti gli esempi sono stati verificati direttamente dal testo, che è in una forma leggibile dalla macchina. Inoltre, nel mostrare l'uso efficace del linguaggio, abbiamo anche cercato, dove possibile, di illustrare algoritmi utili e principi di buona progettazione.
Il libro non è un manuale introduttivo alla programmazione; presuppone un certo grado di familiarità con i concetti basilari della materia come variabili, istruzioni di assegnamento, cicli e funzioni. Ciononostante, un programmatore principiante dovrebbe essere in grado di procedere nella lettura ed acquisire il linguaggio, anche se il consultarsi con un collega più esperto potrà essere di aiuto.
Nella nostra esperienza il C è risultato essere un linguaggio piacevole, espressivo e versatile per una larga varietà di programmi. È facile da imparare e ben si adatta ai vari livelli di esperienza. Noi speriamo che questo libro vi aiuterà a usare il linguaggio al meglio.
Le critiche ed i suggerimenti validi di molti amici e colleghi hanno apportato molto a questo libro ed alla nostra soddisfazione nello scriverlo. In particolare Mike Bianchi, Jim

---

(1) UNIX è un marchio registrato dei Bell Laboratories. Il sistema operativo UNIX è disponibile su licenza della Western Electric, Greensboro, N.C.

Blue, Stu Feldman, Doug McIlroy, Bill Roome, Bob Rosin e Larry Rosler ne hanno letto con molta attenzione numerose versioni. Siamo inoltre debitori e Al Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Haley, Marion Harris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pinson, Bill Plauger, Jerry Spivack, Ken Thompson, e Peter Weinberger per i loro utili commenti durante le varie fasi del libro, e a Mike Lesk e Joe Ossanna per l'inestimabile assistenza nella fotocomposizione.

Brian W. Kernighan

Dennis M. Ritchie

CAPITOLO 0

INTRODUZIONE

Il C è un linguaggio di programmazione di applicazione generale. È stato sempre messo in stretta relazione a UNIX in quanto è stato sviluppato su questo sistema operativo, e poiché UNIX stesso ed il suo software sono scritti in C. Tuttavia il linguaggio non è legato ad alcun sistema operativo o macchina e, nonostante sia stato considerato come un "linguaggio di programmazione di sistema" per la sua utilità nella scrittura di sistemi operativi, è stato usato altrettanto bene per scrivere grossi programmi numerici, di text processing e data-bases.
Il C è un linguaggio relativamente a "basso livello". Questa non è certo una caratteristica negativa; significa semplicemente che il C ha a che fare con lo stesso tipo di oggetti trattati dalla maggior parte dei computers come caratteri, numeri ed indirizzi. Questi possono essere combinati e trasferiti attraverso gli operatori logici ed aritmetici implementati dalle macchine attuali.
Il C non dispone di operazioni per trattare direttamente oggetti strutturati come stringhe di caratteri, insiemi, liste ed array considerati come un'unità. Non esistono, ad esempio, operazioni analoghe a quelle del PL/1 che manipolano interi array o stringhe. Il linguaggio non definisce altre possibilità di allocazione di memoria che la definizione statica e la gestione a stack fornita dalle variabili locali delle funzioni: non ci sono meccanismi di heap o "garage collection" come quelli dell'Algol 68. Infine il C in sé stesso non possiede funzioni di input-output: non ci sono istruzioni READ e WRITE, né sono predisposti metodi di accesso ai files. Tutti questi meccanismi di livello più alto devono essere supportati da funzioni chiamate esplicitamente.
Allo stesso odo, il C offre solo lineari costrutti di controllo: test, cicli, raggruppamenti e sottoprogrammi, ma non la multiprogrammazione, operazioni parallele, sincronizzazione e co-routines.
Sebbene l'assenza di alcune di queste caratteristiche possa sembrare una grave mancanza ("Vuoi dire che devo chiamare una funzione per confrontare due stringhe di caratteri ?"), il fatto di aver contenuto il linguaggio in dimensioni modeste ha portato vantaggi reali. Poiché il C è relativamente piccolo, può essere descritto in uno spazio limitato ed imparato rapidamente. Un compilatore per il C può essere semplice e compatto. I compilatori stessi vengono scritti velocemente; usando la tecnologia dei nostri giorni, è probabile che si produca un compilatore per una nuova macchina in un paio di mesi, scoprendo che l'80% del codice del nuovo compilatore è comune a quelli già esistenti. Ciò produce un alto valore di mobilità del linguaggio. Poiché i tipi di dati e le strutture di controllo fornite dal C sono direttamente supportate dalla maggior parte dei computer esistenti, la libreria richiesta in esecuzione per implementare programmi a se stanti, risulta piccola. Sul PDP/11, per esempio, contiene solamente le ruotine per

ottenere la moltiplicazione e la divisione in 32 bit e per gestire le sequenze di entrata e uscita dalle subroutine. Naturalmente, ogni implementazione provvede a dare una comprensibile e compatibile libreria di funzioni per isolare l'I/O, la gestione delle stringhe e le operazioni di allocazione di memoria, ma poiché vengono chiamate solo esplicitamente, possono essere evitate se opportuno; anch'esse possono essere scritte portabilmente in C.

Poiché il linguaggio riflette le prestazioni dei computer attuali, i programmi C tendono ad essere abbastanza efficienti da non avere l'esigenza di essere riscritti in linguaggio assembler. Il più ovvio esempio è il sistema operativo UNIX stesso, scritto quasi interamente in C. Su 13000 linee di codice di sistema, solo appena 800 linee al livello più basso sono in assembler. Inoltre, praticamente tutti i programmi applicativi di UNIX sono scritti in C; la stragrande maggioranza degli utenti UNIX (compreso uno degli autori di questo libro) non conoscono neanche l'assembler del PDP-11.

Sebbene il C si sposi con le capacità di molti computer, è indipendente da qualsiasi particolare architettura di macchina, per cui con un piccolo sforzo è facile scrivere programmi "portabili", cioè programmi che possono essere eseguiti senza cambiamenti su svariati hardware. Nel nostro ambiente è normale che il software prodotto su UNIX venga trasportato sui sistemi locali Honeywell, IBM e Interdata. Infatti, i compilatori C e il supporto in run-time di queste quattro macchine sono molto più compatibili delle versioni sulla carta standard ANSI del Fortran. Il sistema operativo UNIX stesso gira ora sia sul PDP-11 che su Interdata 8/32. Al di fuori dei programmi che sono necessariamente in un certo modo dipendenti dalla macchina come il compilatore, l'assemblatore ed il debugger, il software scritto in C è identico su entrambe le macchine. All'interno del sistema operativo stesso, le 7000 linee di codice al di fuori del supporto dell'assembler e della gestione dell'I/O sui device, sono identiche per circa il 95%.

Per i programmatori che hanno familiarità con altri linguaggi, menzionare alcuni aspetti storici, tecnici e filosofici del C può risultare utile per fare raffronti.

Molte delle idee più importanti del C discendono dal certamente vecchio ma tuttora piuttosto vitale linguaggio BCPL, sviluppato da Martin Richards. L'influenza di BCPL sul C procedette indirettamente attraverso il linguaggio B, scritto da Ken Thompson nel 1970 per il primo sistema UNIX sul PDP-7.

Sebbene condivida molte caratteristiche con BCPL, C non ne è in nessun senso un dialetto. BCPL e B sono linguaggi "senza tipi": l'unico tipo di dati è la parola-macchina e l'accesso ad altri tipi di oggetti viene ottenuto attraverso speciali operatori o chiamate di funzione. Nel C gli oggetti fondamentali sono caratteri, interi di varie grandezze e numero in virgola mobile. In più, esiste una gerarchia di tipi di dati derivati creati con puntatori, array, strutture, unioni e funzioni.

Il C fornisce le fondamentali costruzioni di controllo del flusso richieste per programmi ben strutturati: raggruppamento di istruzioni; decisioni (if); cicli con il test di uscita in alto (while, for) o in basso (do); selezione di uno in un insieme di possibili casi (switch). (Tutto ciò è stato ugualmente fornito da BCPL, con alcune differenze sintattiche; quel linguaggio ha anticipato la voga della "programmazione strutturata" di parecchi anni).

Il C dispone di puntatori e della capacità di ottenere un'aritmetica sugli indirizzi. Gli argomenti delle funzioni vengono passati copiando il valore dell'argomento e per la funzione stessa è impossibile cambiare l'argomento vero e proprio nel chiamante. Quando si vuole avere una "chiamata per indirizzo", si può passare un puntatore esplicitamente e la funzione può così cambiare l'oggetto a cui punta il puntatore. I nomi

degli array vengono passati come la locazione dell'origine dell'array, per cui gli argomenti di tipo array vengono effettivamente chiamati per indirizzo.
Ogni funzione può essere chiamata ricorsivamente e le sue variabili locali sono tipicamente "automatiche", o create nuovamente ad ogni invocazione. Le definizioni di funzione non possono essere nidificate ma le variabili possono essere dichiarate con uno stile strutturato a blocchi. Le funzioni di un programma C possono essere compilate separatamente. Le variabili possono essere interne ad una funzione, esterne ma conosciute solo all'interno di un singolo file sorgente, o completamente globali. Le variabili interne possono essere automatiche o statiche. Le variabili automatiche possono essere collocate nei registri per aumentarne l'efficienza sebbene la dichiarazione dei registri è solo un suggerimento per il compilatore e non si riferisce agli specifici registri della macchina.
C non è un linguaggio di scrittura rigorosa come il Pascal o l'Algol 68. È relativamente permissivo sulle conversioni di dati, sebbene non converte automaticamente tipi di dati in maniera incontrollata come il PL/I. I compilatori esistenti non supportano controlli in esecuzione per gli indici degli array, tipi di argomenti, ecc.
In situazioni in cui si vuole un forte controllo sui tipi, viene usata una separata versione del compilatore. Questo programma è chiamato *lint* poiché apparentemente scova parti scorrette in un programma. *lint* non genera codice, ma applica uno stretto controllo a molti aspetti di un programma che non possono essere verificati nelle fasi di compilazione e di caricamento. Analizza i conflitti di tipo, l'uso incoerente degli argomenti, variabili inutilizzate o apparentemente non inizializzate, potenziali difficoltà di portabilità e cose simili. I programmi che passano indenni attraverso *lint* godono, a parte poche eccezioni, la completa libertà da errori sui tipi come ad esempio i programmi Algol 68. Menzioneremo altre caratteristiche di *lint* quando se ne presenterà l'occasione.
Infine il C, come ogni altro linguaggio, ha le sue imperfezioni. Alcuni operatori hanno la priorità sbagliata; alcune parti sintattiche potrebbero essere migliori; ci sono varie versioni esistenti del linguaggio che si differenziano per piccole cose. Comunque, il C si è dimostrato essere un linguaggio estremamente efficace ed espressivo per una grande varietà di applicazioni.
Il resto del libro è organizzato come segue. Il Capitolo 1 è un'introduzione didattica alla parte centrale del C. Lo scopo è di fare in modo che il lettore inizi il più velocemente possibile, poiché crediamo fortemente che l'unica materia per imparare un nuovo linguaggio è quella di scriverci programmi. L'approccio presuppone una certa conoscenza degli elementi basilari della programmazione; non c'è alcuna spiegazione sui computer, sulla compilazione, sul significato di un'espressione come n=n+1. Anche se dove possibile abbiamo tentato di illustrare utili tecniche di programmazione, questo libro non intende essere un punto di riferimento sugli algoritmi e le strutture di dati; quando siamo stati forzati a una scelta, ci siamo soffermati sul linguaggio.
I Capitoli dal 2 al 6 trattano vari aspetti del C più in dettaglio e anche più formalmente di quanto lo faccia il Capitolo 1 anche se l'attenzione è sempre su esempi di programmi completi e utili piuttosto di frammenti isolati. Il Capitolo 2 tratta i tipi basilari dei dati, operatori ed espressioni. Il Capitolo 3 tratta i flussi di controllo: if-else, while, for, ecc... Il Capitolo 4 analizza la struttura delel funzioni e del programma — variabili esterne, regole di visibilità, e così via. Il Capitolo 5 discute i puntatori e l'aritmetica degli indirizzi. Il Capitolo 6 contiene i dettagli delle strutture e delle unioni.
Il Capitolo 7 descrive la libreria standard C per l'I/O, che costituisce una comune interfaccia col sistema operativo. Questa libreria di I/O viene supportata da tutte le

macchine che supportano il C, così i programmi che la usano per l'input, l'output ed altre funzioni di sistema possono essere spostati da un sistema all'altro praticamente senza cambiamenti.
Il Capitolo 8 descrive l'interfaccia tra i programmi C e il sistema operativo UNIX, concentrandosi sull'input/output, il file system e la portabilità. Anche se alcuni di questi capitoli sono specifici su UNIX, i programmatori che non stanno usando un sistema UNIX possono qui ugualmente trovare materiale utile, comprendente alcune osservazioni su come è implementata una versione della libreria stadard, e suggerimenti su come ottenere un codice portabile.
L'Appendice A contiene il manuale di riferimento del C. Questa è la voce "ufficiale" sulla sintassi e semantica del C e (eccetto per un proprio compilatore) il guidice finale di ogni omissione e ambiguità dei precedenti capitoli.
Poiché il C è un linguaggio in evoluzione che esiste su una varietà di sistemi, può darsi che qualche parte di questo libro non corrisponda alla situazione corrente dell'implementazione su un dato sistema. Abbiamo tentato di trattare chiaramente tali problemi e di avvertire circa le potenziali difficoltà. In casi dubbi, comunque, abbiamo generalmente scelto di descrivere la situazione del PDP-11 UNIX poiché questo è l'ambiente della maggioranza dei programmatori C. Inoltre l'Appendice A descrive le differenze tra i più importanti sistemi C.

CAPITOLO 1

UN'INTRODUZIONE DIDATTICA

Iniziamo con una veloce introduzione al C. Il nostro scopo è quello di illustrare gli elementi essenziali del linguaggio in veri programmi, senza entrare in dettagli, regole formali ed eccezioni. A questo punto non stiamo cercando di essere né completi né precisi, fermo restando che gli esempi sono da intendersi corretti. È nostra intenzione portarvi il più velocemente possibile al punto in cui possiate scrivere programmi utili e per fare ciò dobbiamo concentrarci sui fondamenti: variabili e costanti, aritmetica, strutture di controllo, funzioni ed i rudimenti di input/output. Lasciamo fuori intenzionalmente da questo capitolo quelle caratteristiche del C che sono di vitale importanza per scrivere programmi più grossi. Queste includono puntatori, strutture, la maggior parte del ricco insieme di operatori, diverse istruzioni di controllo e una miriade di altri dettagli.
Questo approccio, naturalmente, ha i suoi svantaggi. Il più appariscente è che non si può trovare in una singola parte la trattazione completa di una particolare caratteristica del linguaggio ed il primo approccio, essendo sintetico, può anche ingannare. Poiché con gli esempi non possiamo usare tutta la potenza del C, essi non potranno essere sintetici ed eleganti come dovrebbero essere. Abbiamo tentato di minimizzare questi effetti, ma siate consci della loro esistenza.
Un altro inconveniente è che i successivi capitoli necessariamente ripeteranno gli argomenti trattati in questo capitolo. Speriamo che le ripetizioni siano un fattore più di aiuto che di noia.
Ad ogni modo chi ha esperienza di programmazione sarà in grado di estrapolare ciò che gli serve dal materiale presente in questo capitolo. Chi invece inizia può integrare scrivendo programmi piccoli e simili di propria iniziativa. Entrambi possono utilizzarlo come un modello di riferimento per entrare in descrizioni più dettagliate che iniziano dal Capitolo 2.

1.1 Per cominciare

L'unica maniera per apprendere un nuovo linguaggio di programmazione è scrivere programmi. Il primo programma da scrivere è identito per tutti i linguaggi:

```
Stampa le parole
        salve, mondo
```

Questo è il primo ostacolo; per scavalcarlo bisogna essere in grado di creare il testo del programma, compilarlo correttamente, caricarlo, eseguirlo e sapere dove finisce

l'output. Dopo essersi impossessati di questi dettagli meccanici, tutto il resto è relativamente facile.
In C, il programma per scrivere "salve, mondo" è

```
main()
{
    printf("salve, mondo\n");
}
```

Il modo di eseguire questo programma dipende dal sistema che si sta usando. Come esempio specifico, sul sistema operativo UNIX bisogna creare il programma sorgente in un file il cui nome finisce in "**.c**", come *salve.c* e compilarlo attraverso il comando

cc salve.c

Se non si è tralasciato niente, come l'omissione di un carattere o un errore di battitura, la compilazione procederà silenziosamente creando un file eseguibile chiamato *a.out*. Eseguendolo tramite il comando

a.out

produrrà

```
salve, mondo
```

come output. In altri sistemi le regole potranno essere diverse; consultate in questo caso un esperto.

Esercizio 1-1. Eseguire il programma sul proprio sistema. Provare a tralasciare parti del programma osservando i messaggi di errore che vengono prodotti.

Diamo ora qualche spiegazione sul programma. Un programma in C, qualunque sia la sua grandezza, consiste di una o più funzioni che specificano le operazioni di calcolo che devono essere eseguite. Le funzioni del C sono simili alle funzioni e subroutine di un programma Fortran o alle procedure del **PL/1**, Pascal, ecc. Nel nostro esempio, **main** è una specie di funzione. Normalmente si è liberi di dare alle funzioni un nome a piacere, ma **main** è un nome speciale — i programmi cominciano l'esecuzione all'inizio di **main**. Ciò significa che ogni programma *deve* avere **main** da qualche parte. Generalmente **main** chiamerà altre funzioni per portare a termine il suo lavoro; alcune saranno all'interno dello stesso programma, altre risiederanno in librerie di funzioni precedentemente scritte.
Un metodo per comunicare dati tra le funzioni è rappresentato dagli argomenti. Le parentesi che seguono le funzioni contengono la lista di argomenti; in questo caso main è una funzione senza argomenti, indicata con (). Le graffe { } includono le istruzioni che costituiscono la funzione; sono analoghe a **DO-END** del **PL/1**, al **begin-end** di Algol, Pascal e così via. Una funzione viene chiamata nominalmente seguita dalla lista di argomenti tra parentesi.

Non esiste l'istruzione **CALL** come in Fortran o **PL/1**. Le parentesi devono essere presenti anche se non ci sono argomenti.
La linea

```
printf("salve, mondo\n");
```

è una chiamata di funzione il cui nome è **printf** ed il cui argomento è "**salve mondo\n**". **printf** è una funzione di libreria che stampa l'output su terminale (a meno che non sia specificata un'altra destinazione). In questo caso stampa la stringa di caratteri che costituisce il proprio argomento.
Una sequenza di qualsiasi numero di caratteri racchiusa tra doppi apici "..." è chiamata "*stringa di caratteri*" o "*stringa costante*". Per ora l'unico uso delle stringhe di caratteri sarà come argomento per **printf** ed altre funzioni.
La sequenza **n** nella stringa è la notazione del C per il carattere di newline che, stampato, avanza il cursore del terminale sul margine sinistro della linea successiva. Se si tralascia **n** (un valido esperimento) si constaterà che l'output non viene terminato da un line feed. L'unica maniera per avere un carattere di newline all'interno di **printf** è con **n**: se si cerca di fare qualcosa come

```
printf("salve, mondo
");
```

il compilatore C stamperà una spiacevole diagnostica circa i doppi apici mancanti.
printf non rende mai un newline automaticamente per cui in questo modo si possono usare più chiamate per costruire una linea di output. Il nostro primo programma avrebbe potuto essere scritto

```
main()
{
    printf("salve, ");
    printf("mondo");
    printf("\n");
}
```

producendo un'output identico.
Da notare che **n** rappresenta un solo carattere. Una *sequenza di escape* come **n** fornisce un meccanismo generale ed estensibile per rappresentare caratteri non visualizzabili. Tra gli altri forniti dal C vi sono **t** per il tab (carattere di tabulazione), **b** per il backspace, \" per i doppi apici e \\ per il backslash stesso.

Esercizio 1-2. Provare a vedere cosa succede quando la stringa di argomento di **printf** contiene **x**, in cui **x** non è un carattere sopra elencato.

1.2 Variabili e Aritmetica

Il prossimo programma stampa la seguente tabella di temperature Fahrenheit ed i loro centigradi o Celsius equivalenti usando la formula E = (5/9) (F-32).

```
  0  -17.8
 20   -6.7
 40    4.4
 60   15.6
...    ...
260  126.7
280  137.8
300  148.9
```

Questo è il programma

```
/* stampa la tabella Fahrenheit-Celsius
     for f = 0, 20, ..., 300  */
main()
{
    int minimo, massimo, passo;
    float fahr, celsius;

    minimo = 0;     /* limite minimo della tabella delle temperature */
    massimo = 300;  /* massimo limite */
    passo = 20;     /* ampiezza del passo */

    fahr = minimo;
    while (fahr <= massimo) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%4.0f %6.1f\n", fahr, celsius);
        fahr = fahr + passo;
    }
}
```

Le prime due linee

```
/* stampa la tabella Fahrenheit-Celsius
     for f = 0, 20, ..., 300  */
```

sono un *commento*, in questo caso di breve spiegazione su cosa fa il programma. Qualsiasi carattere tra /* e */ viene ignorato dal compilatore; possono essere usati liberamente per rendere il programma più facile da capire. I commenti possono comparire ovunque ci sia un blank o un newline.
In C *ogni* variabile deve essere dichiarata prima dell'uso, generalmente all'inizio della funzione prima delle istruzioni. Se viene tralasciata una dichiarazione, si avrà una diagnostica da parte del compilatore. Una dichiarazione consiste in un *tipo* e una lista di variabili di quel tipo, come in

```
int minimo, massimo, passo;
float fahr, celsius;
```

Il tipo **int** sottintende che le variabili elencate siano di tipo *intero*; **float** sta per *floating point* (virgola mobile), numeri che hanno una parte decimale. La precisione di ambedue **int** e **float** dipende dalla macchina che si sta usando. Sul **PDP-11**, per esempio, un **int** è un numero con segno allocato in 16 bit con un intervallo di valori tra -32768 e +32767. Un **float** è un numero in 32 bit la cui precisione ammonta a circa 7 cifre significative in un intervallo da 10^{-38} a 10^{38}. Il Capitolo 2 elenca le dimensioni per altre macchine.
Il C fornisce parecchi altri tipi di dati oltre a **int** e **float**:

```
char     carattere - un singolo byte
short    intero corto
long     intero lungo
double   floating point in doppia precisione
```

Anche le dimensioni di questi oggetti variano da macchina a macchina; i dettagli si trovano nel Capitolo 2. Ci sono anche *array, strutture* ed *unioni* di questi tipi basilari, *puntatori* ad essi e *funzioni* che li ritornano; tutto ciò verrà trattato a tempo debito.
I primi calcoli nel programma di conversione delle temperature iniziano con gli assegnamenti

```
minimo = 0;
massimo = 300;
passo = 20;
fahr = minimo;
```

che assegnano alle variabili un valore iniziale. Le singole istruzioni vengono terminate da punto e virgola.
Ogni linea della tabella viene calcolata nello stesso modo, per cui viene utilizzato un ciclo che si ripete ad ogni linea; questo è lo scopo dell'istruzione **while**

```
while (fahr <= massimo) {
    ...
}
```

Viene esaminata la condizione tra parentesi. Se è vera (**fahr** è minore o uguale di **massimo**,) viene eseguito il corpo del ciclo (tutte le istruzioni incluse tra le graffe { e }). Dopo di che viene riesaminata la condizione e, se vera, il corpo viene rieseguito. Quando la condizione diventa falsa (**fahr** supera **massimo**) il ciclo finisce e l'esecuzione continua sull'istruzione che segue il ciclo. Poiché in questo caso non vi sono altre istruzioni, il programma termina.
Il corpo di un **while** può essere costituito da una o più istruzioni racchiuse tra graffe, come nel convertitore di temperature, o da una sola istruzione come in

```
while (i < j)
    i = 2 * i;
```

In entrambi i casi le istruzioni controllate da **while** sono indentate di un tab per rendersi conto con un'occhiata di quali sono le istruzioni interne al ciclo. L'indentazione mette in evidenza la struttura logica del programma. Sebbene il C sia piuttosto elastico sulla dislocazione delle istruzioni, per la chiarezza di lettura del programma diventano fondamentali un'appropriata indentazione e l'uso degli spazi. Raccomandiamo di scrivere solamente un'istruzione per linea e (generalmente) di lasciare spazi tra gli operatori. La posizione delle graffe non è fondamentale; abbiamo scelto un tra i tanti stili. Scegliete uno stile che vi soddisfi ed utilizzatelo coerentemente.
La maggior parte dei lavori vengono svolti nel corpo del ciclo. La temperatura Celsius viene calcolata ed assegnata a **celsius** attraverso l'istruzione

```
celsius = (5.0/9.0) * (fahr-32.0);
```

La regione per usare 5.0/9.0 piuttosto che più semplicemente 5/9 è che il C, come molti altri linguaggi, *tronca* la divisione intera ignorando le parti decimali. In questo modo 5/9 risulterebbe zero così come il valore di tutte le temperature. Un punto decimale in una costante indica che è un floating point, così 5.0/9.0 è 0.555... che è esattamente ciò che volevamo.
Inoltre abbiamo scritto **32.0** al posto di **32** benché, dato che **fahr** è di tipo **float**, 32 sarebbe stato automaticamente convertito in **float** (in 32.0) prima della sottrazione. Da un punto di vista stilistico è consigliabile scrivere le costanti in floating point mettendo il punto decimale anche se il loro valore è intero; ciò sottolinea a chi legge la loro natura di floating point ed il compilatore vedrà le cose allo stesso modo.
Regole dettagliate su quando gli interi vengono convertiti in floating point si trovano nel Capitolo 2. Per ora ricordiamo che l'assegnamento

```
fahr = minimo
```

ed il test

```
while (fahr <= massimo)
```

funzionano entrambi come ci si aspetta — **int** viene convertito in **float** prima che l'operazione sia eseguita.
Inoltre questo esempio espone maggiormente il funzionamento di **printf**. **printf** è in realtà una funzione di conversione dal formato di generale applicazione, che descriveremo completamente nel Capitolo 7. Il suo primo argomento è una stringa di caratteri da stampare, in cui ogni segno % indica dove deve essere sostituito uno degli altri (secondo, terzo) e in che forma deve essere stampato. Per esempio, nell'istruzione

```
printf("%4.0f %6.1f\n", fahr, celsius);
```

la specifica di conversione **%4.0f** indica che deve essere stampato un numero in floating point in uno spazio di almeno quattro caratteri senza alcuna cifra dopo il punto decimale. **%6.1f** indica che un altro numero occupa almeno sei spazi con una cifra dopo il punto decimale analogamente a **F6.1** del Fortran o **F(6,1)** del **PL/1**. Certe parti di una specifica possono essere omesse: **%6f** indica che il numero deve occupare almeno sei spazi;

%.2f richiede due spazi dopo il punto decimale, ma la lunghezza non è fissata. **%f** indica semplicemente di stampare il numero come floating point. Inoltre **printf** riconosce **%d** per gli interi decimali, **%o** per numeri ottali, **%x** per numeri esadecimali, **%c** per i caratteri, **%s** per stringhe di caratteri e **%%** per lo stesso **%**.
Ogni costruzione **%** nel primo argomento di **printf** è accoppiata con il corrispondente secondo, terzo argomento; se non si vogliono ottenere risultati senza senso, devono susseguirsi correttamente per numero e tipo.
Per inciso, **printf** *non* fa parte del linguaggio C; non esistono funzioni di input o output definite all'interno del C. Non c'è niente di speciale circa **printf**; è semplicemente una funzione di utilità che fa parte della libreria standard di routines che sono normalmente accessibili ai programmi C. Poiché vogliamo concentrarci sul C stesso, non discuteremo a lungo di I/0 fino al Capitolo 7. In particolare, fino ad allora non prenderemo in considerazione l'input con formato. Se bisogna acquisire numeri in input si legga la trattazione della funzione **scanf** nel Capitolo 7 paragrafo 7.4. **scanf** è molto simile a **printf** senonché legge input invece di scrivere output.

Esercizio 1-3. Modificare il programma di conversione delle temperature stampando un'intestazione prima della tabella.

Esercizio 1-4. Scrivere un programma che stampi la tabella corrispondente Celsius-Fahrenheit.

1.3 L'istruzione for

Come vi aspetterete vi è una gran quantità di modi diversi per scrivere un programma; vediamo come esempio una variazione al convertitore di temperature.

```
main()      /* Tabella Fahrenheit-Celsius */
{
    int fahr;

    for (fahr=0; fahr <= 300; fahr = fahr + 20)
        printf("%4d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Questo produce gli stessi risultati, ma ha senz'altro un diverso aspetto. Il cambiamento maggiore è l'eliminazione della maggior parte delle variabili; rimane solamente **fahr** come **int** (per evidenziare la conversione **%d** in **printf**).

I limiti superiori ed inferiori e l'ampiezza del passo appaiono soltanto come costanti nell'istruzione **for**, essa stessa un nuovo modello, e l'espressione che calcola la temperatura Celsius appare ora come terzo argomento di **printf** invece che in una distinta istruzione di assegnamento.
L'ultimo cambiamento è la comparsa di una regola generale del C — in qualsiasi contesto in cui sia possibile usare il valore di una variabile di qualche tipo, si può usare una espressione di quel tipo. Poiché il terzo argomento di **printf** dev'essere un valore in floating point in accordo con **%6. 1f**, può apparire qui una qualsiasi espressione in floating point.
Il **for** stesso è un ciclo, una generalizzazione del **while**. Se ora lo si paragona al precedente **while**, dovrebbe risultare chiara la sua funzione. Contiene tre parti, separate da punti e virgola. La prima parte

```
fahr = 0
```

viene eseguita un'unica volta prima dell'entrata nel ciclo. La seconda parte è il test o condizione che controlla il ciclo:

```
fahr <= 300
```

Questa condizione viene valutata; se è vera il corpo del ciclo (in questo caso un solo **printf**) viene eseguito. Segue il passo di reinizializzazione

```
fahr = fahr + 20
```

e la condizione viene ri-valutata. Il ciclo termina quando la condizione risulta falsa. Come per **while**, il corpo del ciclo può essere una sola istruzione od un gruppo di istruzioni racchiuse tra graffe. Le parti di inizializzazione e reinizializzazione possono essere qualsiasi espressione.
La scelta tra **while** e **for** è libera, generalmente in base alla maggior chiarezza. Il **for** è solitamente adatto per cicli in cui inizializzazione e reinizializzazione sono singole istruzioni correlate logicamente dato che è più compatto di **while** e mantiene unite le istruzioni di controllo del ciclo.

Esercizio 1-5. Modificare il programma di conversione delle temperature per stampare la tavola in ordine inverso, cioè da 300 gradi a 0.

1.4 Costanti simboliche

Un'osservazione finale prima di abbandonare definitivamente la conversione delle temperature. Non è consigliabile inserire in un programma numeri come 300 e 20; non comunicano molte informazioni a chi dovrà leggere il programma e sono difficili da cambiare in maniera sistematica. Il C fortunatamente prevede un modo per evitare questi numeri. All'inizio di un programma si può stabilire attraverso la costruzione # **define** che un *nome simbolico* o *costante simbolica* consistano di una certa stringa di caratteri.
Successivamente il compilatore rimpiazzerà tutte le occorrenze del nome che non si

trovano tra apici con la stringa corrispondente. Il rimpiazzamento per il nome può essere ogni tipo di testo, senza essere limitato ai numeri.

```
#define   MINIMO  0      /* limite inferiore della tabella */
#define   MASSIMO 300    /* limite superiore */
#define   PASSO   20     /* ampiezza del passo */

main()        /* Tabella Fahrenheit-Celsius */
{
    int fahr;

    for (fahr = MINIMO; fahr <= MASSIMO; fahr = fahr + PASSO)
        printf("%4d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Le quantità **MINIMO, MASSIMO e PASSO** sono costanti, per cui non compaiono nelle dichiarazioni. Generalmente i nomi simbolici vengono scritti in maiuscolo per distinguerli dai nomi delle variabili scritti in minuscolo. Da notare l'assenza del punto e virgola alla fine delle definizioni. Siccome viene sostituita l'intera linea dopo il nome definito, nel **for** risulterebbero troppi punti e virgola.

1.5 Una raccolta di programmi utili

Consideriamo ora una famiglia di programmi in relazione tra di loro che effettuano semplici operazioni su dati di tipo carattere. Noterete che molti programmi non sono altro che versioni espanse dei prototipi che trattiamo ora.

Input ed output di caratteri
La libreria standard fornisce funzioni per leggere e scrivere un carattere per volta. **getchar ()** accetta *il prossimo carattere di input* ogni volta che viene chiamata e ritorna il carattere col proprio valore. Per cui, dopo

```
c = getchar()
```

la variabile **c** conterrà il prossimo carattere dell'input. Normalmente i caratteri provengono dal terminale, ma ciò non ci riguarda fino al Capitolo 7.
La funzione **putchar (c)** è il complemento di **getchar**:

```
putchar(c)
```

stampa il contenuto della variabile **c** attraverso qualche strumento di output, anch'esso di solito il terminale. Si possono alternare chiamate a **putchar** e a **printf**; l'output apparirà nello stesso ordine delle chiamate.
Come per **printf**, non c'è niente di speciale circa **getchar** e **putchar**. Non fanno parte del linguaggio C ma sono universalmente disponibili.

Copia di files

Introdotti **getchar** e **putchar**, è possibile scrivere una sorprendente mole di codice utile senza avere ulteriori conoscenze di I/0. L'esempio più semplice è un programma che ricopia l'input sull'output un carattere per volta. Schematicamente,

accetta un carattere,
while (il carattere non è il segnale di fine file)
    manda in output il carattere appena letto
    accetta un altro carattere

Convertendolo in C si ha

```
main()  /* copia l'input sull'output; la versione */
{
      int c;

      c = getchar();
      while (c != EOF) {
          putchar(c);
          c = getchar();
      }
}
```

L'operatore di relazione != significa "diverso da".
Il problema principale è scoprire la fine dell'input. Per convenzione quando **getchar** incontra la fine dell'input ritorna un valore che non rappresenta nessun carattere possibile; in questo modo i programmi possono rendersi conto della fine dell'input. Il problema principale è che esistono due convenzioni di uso comune sul valore di fine file. Abbiamo ignorato il problema usando il nome simbolico **EOF** al posto del valore, qualunque esso sia. In pratica **EOF** può essere -1 oppure 0, per cui il programma deve essere preceduto dall'appropriata definizione.

```
#define EOF -1
```

oppure

```
#define EOF 0
```

per lavorare correttamente. Usando la costante simbolica **EOF** per rappresentare il valore ritornato da **getchar** quando incontra la fine del file, ci siamo assicurati che nel programma solo una cosa dipenda dallo specifico valore numerico.
Inoltre abbiamo dichiarato **c** come **int** e non come **char** in maniera tale che possa rappresentare il valore di ritorno di **getchar**. Come vedremo nel Capitolo 2 questo valore è propriamente **int** poiché deve essere in grado di rappresentare **EOF** oltre a tutti i possibili caratteri.

Il programma di copia avrebbe certamente potuto essere scritto più sinteticamente da esperti programmatori C. In C ogni assegnamento del tipo di

```
c = getchar()
```

può essere usato in un'espressione; il suo valore è semplicemente il valore che viene assegnato alla parte sinistra. Se l'assegnamento di un carattere a **c** viene collocato all'interno della parte di test di un **while**, il programma di copia di file può essere scritto

```
main()  /* copia l'input sull'output; 2a versione */
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

Il programma legge un carattere, lo assegna a **c** e verifica se il carattere è il segnale di fine file. Se non lo è viene eseguito il corpo del **while** stampando il carattere. Dopo di che si ripete il **while**. Alla fine, quando giunge la fine dell'input, il **while** termina insieme a **main**.
Questa versione accentra l'input — c'è ora un'unica chiamata a **getchar** — e sintetizza il programma. Nidificare un assegnamento all'interno di un test è uno dei modi con cui il C permette una preziosa sinteticità (è possibile ignorare tutto ciò creando programmi illeggibili — tendenza che tenteremo di frenare).
È importante prendere nota del fatto che le parentesi attorno all'assegnamento nella espressione condizionale sono necessarie. La ***priorità*** di != è più alta di quella di =, il che significa che in assenza di parentesi il test di relazione != verrebbe eseguito prima dell'assegnamento =.

Così l'istruzione

```
c = getchar() != EOF
```

è equivalente a

```
c = (getchar() != EOF)
```

che ha l'effetto indesiderato di assegnare 0 od 1 a c, a seconda che la chiamata a **getchar** abbia incontrato o meno la fine del file. (Tutto questo verrà approfondito nel Capitolo 2).

Conteggio di caratteri

Il prossimo programma conta i caratteri; è una piccola elaborazione del programma di copia.

```
main()   /* conta i caratteri in input */
{
    long nc;

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

L'istruzione

```
++nc;
```

presenta un nuovo operatore, ++, il cui significato è *incrementa di uno*. Si avrebbe potuto scrivere **nc = nc +1** però **++nc** è più conciso e spesso più efficiente. Esiste il corrispondente operatore **--** per decrementare di uno. Gli operatori ++ e -- possono essere sia operatori prefissi **(++nc)** che postfissi **(nc++)**; queste due forme hanno un differente valore nelle espressioni, come vedremo meglio nel Capitolo 2, ma entrambi **++nc** e **nc++** incrementano **nc**. Per il momento limitiamoci ai prefissi.
Il programma che conta caratteri accumula il conteggio in una variabile di tipo **long** invece di **int**. Sul **PDP-11** il valore massimo di un **int** è **32767** per cui sarebbe sufficiente un input relativamente piccolo per mandare il contatore in overflow (fuori dai propri limiti) se fosse stato dichiarato **int**; sul C di Honeywell e **IBM, long** e **int** sono sinonimi e comunque parecchio più grandi. La conversione specificata da **%ld** segnala a **printf** che l'argomento corrispondente è un intero **long**.
Per far fronte a numeri ancora più grossi, si potrà usare un **double (float** in doppia lunghezza). Inoltre usiamo l'istruzione **for** invece di **while** per illustrare una maniera alternativa di scrivere il ciclo.

```
main()  /* conta i caratteri in input  */
{
    double nc;

    for (nc = 0; getchar() != EOF; ++nc)
            ;
    printf("%.0f\n", nc);
}
```

printf usa **%f** sia per **float** che per **double**; **%.0f** sopprime la stampa della inesistente parte decimale.
In questo caso il corpo del ciclo del **for** è *vuoto* poiché tutto il lavoro viene svolto nelle parti di test e reinizializzazione. Però le regole sintattiche del C richiedono che l'istruzione **for** abbia un corpo. Il punto e virgola a se stante, tecnicamente chiamato *istruzione nulla*, serve a soddisfare tale richiesta. È stato collocato su una linea separata per renderlo più visibile.

Prima di abbandonare il programma di conteggio caratteri, osserviamo che se l'input non contiene alcun carattere, i test all'interno di **while** e **for** falliscono alla prima chiamata di **getchar** in maniera tale che il programma produce zero, il risultato corretto. Questa è un'osservazione importante. Una delle caratteristiche positive di **while** e **for** è che verificano le condizioni *all'inizio* del ciclo prima di procedere nel corpo. Se non c'è niente da fare, niente verrà fatto anche se ciò significa non entrare mai nel corpo del ciclo. I programmi devono comportarsi intelligentemente quando si trovano di fronte a input come "nessun carattere". Le istruzioni **while** e **for** aiutano ad assicurare un funzionamento ragionevole alle condizioni limite.

Conteggio di linee

Il prossimo programma conta le *linee* del proprio input. Si assume che le linee di input siano terminate dal carattere di newline che viene diligentemente posto al termine di ogni linea stampata.

```
main()     /* conta le linee in input */
{
     int c, nl;

     nl = 0;
     while ((c = getchar()) != EOF)
          if (c == '\n')
               ++nl;
     printf("%d\n", nl);
}
```

Il corpo del **while** consiste ora di un **if** che di volta in volta controlla l'incremento **++nl**. L'istruzione **if** verifica la condizione tra parentesi e, se vera, esegue l'istruzione (o il gruppo di istruzioni tra graffe) che lo segue. Abbiamo nuovamente indentato per far vedere cosa è controllato da cosa. Il doppio segno di uguale == è la notazione del C per "uguale a" (come .EQ. del Fortran). Questo simbolo viene usato per distinguere il test di uguaglianza dal simbolo = usato negli assegnamenti. Poiché nei programmi C gli assegnamenti sono circa il doppio frequenti dei test di uguaglianza, è appropriato che l'operatore sia lungo la metà.
Un singolo carattere può essere scritto tra singoli apici per produrre un valore equivalente al valore numerico del carattere nel set di caratteri della macchina; questa notazione è chiamata *costante carattere*. Così, per esempio **'A'** è una costante carattere; nel set di caratteri **ASCII** il suo valore è **65**, la rappresentazione interna del carattere **A**. Naturalmente è da preferirsi **'A'** al numero **65**: è chiaro il significato ed è indipendente da particolari set di caratteri.
Le sequenze di escape usate nelle stringhe di caratteri sono valide anche per le costanti carattere, e anche nei test e nelle espressioni, aritmetiche **'\n'** sta per il valore del carattere di newline. È importante prender nota che **'\n'** è un solo carattere e che nelle espressioni è equivalente ad un solo intero; dall'altra parte **"\n"** è una stringa di caratteri che ha la caratteristica di contenere solo un carattere. L'argomento delle stringhe rispetto ai caratteri verrà discusso nel Capitolo 2.

Esercizio 1-6. Scrivere un programma che conti blank, tab e newline

Esercizio 1-7. Scrivere un programma che copi il proprio input sull'output rimpiazzando una stringa di uno o più blank con un'unico blank.

Esercizio 1-8. Scrivere un programma che rimpiazzi ogni tab con la sequenza di tre caratteri >, *backspace*, -, che viene stampato come →, ed ogni backspace dalla simile sequenza ←. Ciò rende visibili i tab ed i backspace.

Conteggio di parole

Il quarto della nostra serie di programmi di utilità conta linee, parole e caratteri, con l'approssimativa definizione che una parola è qualsiasi sequenza di caratteri, con l'approssimativa definizione che una parola è qualsiasi sequenza di caratteri che non contenga blank, tab o newline (è una semplice versione del programma di utilità **UNIX** *wc*).

```
#define    SI   1
#define    NO   0

main()     /* conta linee, parole e caratteri in input */
{
    int c, nl, np, nc, inparola;

    inparola = NO;
    nl = np = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            inparola = NO;
        else if (inparola == NO)  {
            inparola = SI;
            ++np;
        }
    }
    printf("%d %d %d\n", nl, np, nc);
}
```

Ogni volta che il programma trova il primo carattere di una parola, la conta. La variabile *inparola* segnala se il programma sia o meno in quel momento su una parola; inizialmente è "non in parola", a cui viene assegnato il valore **NO**. Preferiamo le costanti simboliche **SI** e **NO** ai valori **1** e **0** poiché rendono il programma più leggibile. Naturalmente in un programma così, piccolo non fa molta differenza, ma in programmi più grossi, la chiarezza che si guadagna giustifica certo il piccolo sforzo di scriverlo così dall'inizio.

Si può notare che è più facile fare sostanziali cambiamenti in un programma in cui i numeri appaiono solo come costanti simboliche.
La linea

```
nl = np = nc = 0;
```

assegna zero a tutte e tre le variabili. Questo non è un caso speciale, ma una conseguenza del fatto che un assegnamento ha un valore e gli assegnamenti vengono associati da destra a sinistra. È come se avessimo scritto

```
nc = (nl = (np = 0));
```

L'operatore `||` significa **OR**, così la linea

```
if (c == ' ' || c == '\n' || c == '\t')
```

significa "se **c** è blank *oppure* **c** è newline *oppure* **c** è tab...". (La sequenza di escape **\t** è la rappresentazione visibile del carattere tab). Esiste il corrispondente operatore **&&** per **AND**. Le espressioni connesse da **&&** oppure **||** vengono valutate da sinistra a destra con la certezza che la valutazione si interrompe appena viene stabilita la verità o falsità. Per cui se **c** contiene un blank non c'è bisogno di verificare se contiene newline o tab, e questi test *non* vengono fatti. Ciò non è fondamentale qui ma lo diventa in situazioni più complicate, come vedremo tra poco.
Inoltre l'esempio mostra l'istruzione **else** del C, che specifica un'azione alternativa se la parte condizionale di un'istruzione **if** è falsa. La forma generale è

```
if (espressione)
    istruzione-1
else
    istruzione-2
```

Viene eseguita una ed una sola delle due istruzioni associate ad un **if-else**. Se *espressione* è vera, viene eseguita *istruzione-1*, altrimenti viene eseguita *istruzione-2*. Ognuna può essere anche piuttosto complicata. Nel programma che conta le parole l'istruzione dopo **else** è un **if** che controlla due istruzioni tra graffe.

Esercizio 1-9 Come si potrebbe controllare il programma che conta parole? Quali sono alcuni suoi limiti?

Esercizio 1-10. Scrivere un programma che stampi le parole del suo input, una per linea.

Esercizio 1-11. Modificare il programma che conta le parole usando una definizione migliore di "parola", per esempio una sequenza di lettere, cifre e apostrofi che inizia con una lettera.

1.6 Arrays

Scriviamo ora un programma che conti il numero di occorrenze di ogni cifra, di caratteri di spaziatura (blank, tab e newline) e di tutti gli altri caratteri. Naturalmente ciò è artificiale, ma ci permette di illustrare parecchi aspetti del C in un solo programma.
Vi sono dodici categorie di input, per cui è conveniente usare un array che contenga il numero di occorrenze di ogni cifra piuttosto di dieci variabili semplici. Questa è una versione del programma:

```
main()     /* conta cifre, spazi bianchi, altri */
{
     int c, i, nbianchi, naltri;
     int ncifre[10];

     nbianchi = naltri = 0;
     for (i = 0; i < 10; ++i)
          ncifre[i] = 0;

     while ((c = getchar()) != EOF)
          if (c >= '0' && c <= '9')
               ++ncifre[c-'0'];
          else if (c == ' ' || c == '\n' || c == '\t')
               ++nbianchi;
          else
               ++naltri;

     printf("cifre =");
     for (i = 0; i <10; ++i)
          printf(" %d", ncifre[i]);
     printf("\nspazi bianchi = %d, altri = %d\n",
          nbianchi, naltri);
}
```

La dichiarazione

```
int ncifre[10];
```

dichiara che **ncifre** è un array di 10 interi. In C gli indici degli array partono sempre da zero, (invece che da uno come in Fortran o PL/1). Perciò gli elementi sono **ncifre[0], ncifre[1],..., ncifre[9]**. Ciò si riflette nei cicli **for** che inizializzano e stampano gli array. Un indice può essere un'espressione intera qualunque, che naturalmente può essere composta da variabili intere come **i**, e da costanti intere.
Questo programma mette particolarmente in rilievo le proprietà delle rappresentazioni in caratteri delle cifre. Per esempio, il test

```
if (c >= '0' && c <= '9') ...
```

determina se il carattere **c** è una cifra o meno. Se lo è, il valore numerico della cifra è

```
c - '0'
```

Questo funziona solo se **'0', '1'**, ecc. sono positivi ed in ordine crescente, e se tra **'0'** e **'9'** non esiste altro all'infuori di cifre. Fortunatamente questo è vero per tutti i set di caratteri convenzionali.
Per definizione, le espressioni aritmetiche che contengono tipi **char** e **int**, convertono tutto in **int** prima di procedere, così in contesti aritmetici le variabili o costanti **char** sono identiche ad **int**. Questo risulta comodo e naturale; per esempio, **c - '0'** è un espressione intera con valore tra 0 e 9 corrispondente al carattere tra **'0** e **'9'** contenuto in **c**, e pertanto un valido indice per l'array **ncifre**.
Per determinare se un carattere è una cifra, uno spazio o qualcos'altro, viene usata la sequenza

```
if (c >= '0' && c <= '9')
    ++ncifre[c-'0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++nbianchi;
else
    ++naltri;
```

Il modello

```
if (condizione)
    istruzione
else if (condizione)
    istruzione
else
    istruzione
```

compare spesso nei programmi come un modo per esprimere una decisione a più vie. Il codice viene semplicemente letto dall'inizio finché qualche *condizione* non viene soddisfatta; a quel punto viene eseguita la corrispondente parte di *istruzione* e termina l'intera costruzione. (Naturalmente l'*istruzione* può essere un gruppo di istruzioni tra graffe). Se non è stata soddisfatta alcuna condizione, viene eseguita, se presente, l'*istruzione* dopo l'**else** finale. Se sono omessi l'**else** e *istruzione* finali, (come nel programma di conteggio parole), non viene eseguito niente. Può comparire un numero arbitrario di gruppi

```
else if (condizione)
    istruzione
```

tra l'**if** iniziale e l'**else** finale. Per quanto riguarda lo stile, è consigliabile dare a questa costruzione l'aspetto che abbiamo visto, così le condizioni lunghe non escono dal limite destro della pagina.
L'istruzione **switch**, che sarà discussa nel Capitolo 3, fornisce un'altra maniera per scrivere strutture a più vie che è particolarmente utile quando la condizione che deve essere verificata è semplicemente un'espressione di interi o di caratteri che deve uguagliare una costante che fa parte di un insieme. Per confronto, presenteremo nel Capitolo 3 una versione di questo programma con **switch**.

Esercizio 1-12. Scrivere un programma che stampa un istogramma della lunghezza delle parole del suo input. È più facile disegnare un istogramma orizzontalmente; l'orientamento verticale è più impegnativo.

1.7 Funzioni

In C una *funzione* è equivalente ad una subroutine o funzione nel Fortran o ad una procedura nel **PL/1**, Pascal, ecc. Una funzione costituisce un'utile modo per relegare certi calcoli in una scatola nera, che può essere successivamente usata senza preoccuparsi del suo interno. Le funzioni sono l'unico strumento per far fronte alle potenziali difficoltà di grossi programmi. Sviluppando appropriate funzioni, è possibile ignorare *come* viene svolto un certo lavoro; è sufficiente sapere *cosa* viene fatto. Il C è progettato per fare in modo che l'uso delle funzioni sia facile, conveniente, efficiente; capiterà spesso di vedere funzioni lunghe solo poche righe e chiamate una sola volta, solo perché chiariscono alcune parti di programma.
Fino ad ora abbiamo usato solamente funzioni come **printf, getchar** e **putchar** che sono già a nostra disposizione; è ora di scriverne alcune di propria mano. Poiché il C non ha un operatore di esponenziazione come ****** del Fortran o **PL/1**, vediamo il meccanismo di definizione di funzione scrivendo la funzione **potenza (m, n)** per elevare un intero **m** alla potenza intera positiva ennesima. Ad esempio, il valore di **potenza (2, 5)** è **32**. Certamente questa funzione non svolge l'intero compito di ****** poiché gestisce solo potenze positive di piccoli numeri interi, ma per non confondersi è meglio andare a piccoli passi.
Questa è la funzione **potenza** in un programma principale che la sfrutta, per vedere in una sola volta l'intera struttura.

```
main()     /* test sulla funzione potenza */
{
     int i;

     for (i = 0; i < 10; ++i)
          printf("%d %d %d\n", i, potenza(2,i),potenza(-3,i));
}

potenza(x, n)     /* eleva x alla n-esima potenza; n > 0 */
int x, n;
{
     int i, p;

     p = 1;
     for (i = 1; i <= n; ++i)
          p = p * x;
     return(p);
}
```

Tutte le funzioni hanno la stessa forma

nome (lista di argomenti, se ce ne sono)
dichiarazioni degli argomenti, se ce ne sono
{
    dichiarazioni
    istruzioni
}

Le due funzioni possono apparire in qualsiasi ordine, in uno o due files sorgenti. Naturalmente se il sorgente appare in due files, bisognerà dare più informazioni per compilarlo e caricarlo di quanto si sarebbe fatto se fosse apparso in uno solo, ma questa è una questione che riguarda il sistema operativo, non un attributo del linguaggio. Per il momento assumiamo che entrambe le funzioni risiedano nello stesso file, in modo che qualunque cosa abbiate imparato su come eseguire programmi C non cambi.
La funzione **potenza** viene chiamata due volte nella linea

```
printf("%d %d %d\n", i, potenza(2,i),potenza(-3,i));
```

Ogni chiamata passa due argomenti a **potenza**, la quale ogni volta restituisce un intero a cui viene dato il formato richiesto e stampato. In un'espressione, **potenza (2, i)** è un intero come lo sono **2** ed **i**. (Non tutte le funzioni producono un valore intero; ne parleremo nel Capitolo 4).
In **potenza** gli argomenti devono essere dichiarati in maniera appropriata affinché siano noti i loro tipi. Questo viene fatto nella linea

```
int x, n;
```

che segue il nome della funzione. La dichiarazione degli argomenti si colloca tra la lista di argomenti e l'apertura della graffa; ogni dichiarazione termina con punto e virgola. I nomi usati da **potenza** per i suoi argomenti sono realmente *locali* a **potenza** e non accessibili da nessun'altra funzione: gli stessi nomi possono essere usati da altre routines senza conflitto. Ciò vale anche per le variabili **i** e **p**: la **i** in **potenza** non ha alcuna relazione con la **i** in **main**.
Il valore calcolato da **potenza** viene passato a **main** attraverso l'istruzione **return**, come in **PL/1**. Internamente alle parentesi può comparire qualsiasi espressione. Non è necessario che una funzione produca un valore; l'istruzione **return** senza espressioni restituisce il controllo senza alcun valore al chiamante, così come quando la funzione termina per aver incontrato l'ultima graffa chiusa.

Esercizio 1-13. Scrivere un programma che converta il proprio input in lettere minuscole usando la funzione **lower (c)** che ritorna **c** se **c** non è una lettera ed il valore minuscolo di **c** se è una lettera.

1. 8 Argomenti — Chiamata per Valore

Un aspetto delle funzioni del C può essere poco familiare a programmatori che sono abituati ad altri linguaggi, in particolare Fortran e **PL/1**. In C tutti gli argomenti delle funzioni vengono passati "per valore". Ciò significa che alla funzione chiamata vengono passati i valori dei suoi argomenti in variabili temporanee (residenti nello stack) invece che con il loro indirizzo. Questo porta alcune proprietà differenti dei linguaggi con "chiamata per indirizzo" come Fortran e **PL/1**, in cui alla routine chiamata viene passato l'indirizzo dell'argomento e non il suo valore.

La differenza principale è che in C la funzione chiamata *non può* alterare una variabile della funzione chiamante; può solamente alterare la sua copia privata e temporanea. Comunque la chiamata per valore è un vantaggio, non una restrizione. Generalmente porta a programmi più compatti con meno variabili aggiunte poiché nella routine chiamata gli argomenti possono essere trattati come variabili locali già inizializzate convenientemente. Per esempio, questa versione di **potenza** fa uso di questa caratteristica.

```
potenza(x, n)      /* eleva x alla n-esima potenza; n>0; versione 2 */
int x, n;
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * x;
    return(p);
}
```

L'argomento **n** è usato come variabile temporanea e viene decrementato finché non diventa zero; la variabile **i** non è più necessaria. Qualsiasi operazione su di **n** all'interno di **potenza** non ha alcun effetto sull'argomento con cui **potenza** è stata originalmente chiamata.
In caso di necessità, è possibile fare in modo che una funzione modifichi una variabile della routine chiamante. La (funzione) chiamante deve provvedere a generare l'*indirizzo* della variabile da modificare (tecnicamente un *puntatore* alla variabile), e nella funzione bisogna dichiarare l'argomento come puntatore e riferirsi alla variabile indirettamente attraverso questo. Ciò verrà trattato in dettaglio nel Capitolo 5.
Quando viene usato il nome di un array come argomento, il valore che viene passato alla funzione è effettivamente l'indirizzo o locazione di inizio dell'array (*non* c'è la copia degli elementi dell'array). Indicizzando questo valore, la funzione può accedere ed alterare ogni elemento dell'array. Questo è l'argomento principale del prossimo paragrafo.

1.9 Array di caratteri

Probabilmente il tipo più comune di array in C è l'array di caratteri. Per illustrare l'uso degli array di caratteri e le funzioni per manipolarli, scriviamo un programma che legge un insieme di linee e ne stampa la più lunga. Lo schema di base è alquanto semplice:

while *(c'è un'altra linea)*
    if *(è più lunga della più lunga precedente)*
        salvala con la sua lunghezza
stampa la linea più lunga

Lo schema rende chiaro che il programma si divide naturalmente in parti. Una parte legge una nuova linea, un'altra la confronta, un'altra la salva ed il resto controlla il processo.
Poiché le azioni si dividono in modo così soddisfacente, sarebbe anche bene scriverle

la *successiva linea* di input; questa è una generalizzazione di **getchar**. Per fare in modo che la funzione sia utile in altri contesti, cercheremo di renderla il più flessibile possibile. Come minimo, **getline** deve ritornare un segnale su una possibile fine file; rendendola di utilità più generale, potrebbe ritornare la lunghezza della linea o zero se è stata incontrata la fine dell'input. Zero non è mai una valida lunghezza di una linea poiché ogni linea ha perlomeno un carattere; anche una linea contenente solo un newline ha lunghezza 1.
Quando incontriamo una linea più lunga della più lunga precedente, deve venire salvata da qualche parte. Ciò suggerisce una seconda funzione, **copy**, per copiare la nuova linea in un posto sicuro.
Infine, necessitiamo di un programma principale che controlli **getline** e **copy**. Questo è il risultato.

```
#define MAXLINEE  1000  /* massima lunghezza linea di input */

main()    /* trova la linea piu' lunga */
{
    int  lung; /* lunghezza della linea corrente */
    int  max; /* lunghezza massima finora incontrata */
    char linea[MAXLINEE]; /* linea corrente di input */
    char salva[MAXLINEE]; /* linea piu' lunga, salvata */

    max = 0;
    while ((lung = getline(linea, MAXLINEE)) > 0)
    if (lung > max) {
        max = lung;
        copy(linea, salva);
    }
    if (max > 0)    /* c'era una linea */
    printf("%s", salva);
}

getline(s, lim)     /* legge la linea in s, ne ritorna la lunghezza */
char s[];
int lim;
{
    int c, i;

    for (i=0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return(i);
}

copy(s1, s2)  /* copy s1 to s2; assume s2 abbastanza grande */
char s1[], s2[];
{
    int i;

    i = 0;
    while ((s2[i] = s1[i]) != '\0')
        ++i;
}
```

main e **getline** comunicano tra di loro attraverso una coppia di argomenti ed un valore di ritorno. In **getline** gli argomenti sono dichiarati dalle linee

```
char s[];
int lim;
```

che specificano che il primo argomento è un array ed il secondo un intero. La lunghezza dell'array *s* non è stata specificata in **getline** poiché è già stata determinata in **main**
getline usa **return** per restituire un valore al chiamante, come già faceva la funzione **potenza**. Alcune funzioni ritornano un valore utile; altre, come **copy**, vengono solamente usate per i propri effetti e non ritornano valori.
getline immette il carattere \0 (il *carattere nullo* il cui valore è zero) alla fine dell'array che sta creando per segnare la fine della stringa di caratteri. Questa convenzione viene usata anche dal compilatore C: quando viene scritta una costante stringa come

```
"salve\n"
```

in un programma C, il compilatore crea un array di caratteri contenente i caratteri della stringa e lo termina con un **\0** in maniera tale che funzioni come **printf** possano vederne la fine:

s	a	l	v	e	\n	\0

La specifica di formato **%s** di **printf** si aspetta una stringa rappresentata in questa forma. Se si esamina **copy**, si scoprirà che conta sul fatto che il proprio argomento in input **s1** termini con **\0** e ricopia questo carattere nell'argomento di output **s2**. (Tutto ciò implica che **\0** non deve fare parte di un normale testo).
È doveroso menzionare che anche un programma piccolo come questo presenta qualche problema di difficile soluzione. Per esempio, come si comporta **main** se incontra una linea che eccede i suoi limiti? **getline** lavora correttamente in quanto cessa di accumulare caratteri quando l'array è pieno, anche se non è stato incontrato newline. Verificandone la lunghezza e l'ultimo carattere restituito, main riesce a determinare se la linea fosse troppo lunga e poi agire a suo piacere. Per brevità abbiamo ignorato questo fatto.
Per chi usa **getline** non c'è modo di sapere a priori quanto sarà lunga una linea di input, così **getline** controlla l'eventuale overflow. Dall'altra parte, chi usa **copy** già conosce (o può conoscere) quanto grosse siano le stringhe, così abbiamo deciso di non aggiungergli verifiche di errore.

Esercizio 1-14. Si riveda la routine principale del programma della linea più lunga facendo in modo che stampi correttamente la lunghezza di linee di input di lunghezza arbitraria, ed il più possibile del testo.

Esercizio 1-15. Scrivere un programma che stampi tutte le linee più lunghe di 80 caratteri.

Esercizio 1-16. Scrivere un programma che rimuova il blank ed i tab in coda ad ogni linea e che cancelli linee interamente bianche

Esercizio 1-17. Scrivere la funzione **reverse** (s) la quale inverte la stringa di caratteri **s**. La si usi per scrivere un programma che inverte il proprio input una linea per volta.

1.10 Visibilità; Variabili Esterne

Le variabili in **main (linea, salva,** ecc.) sono private o locali a **main**; siccome sono dichiarate insieme a main, nessun'altra funzione può avere accesso diretto ad esse. Lo stesso vale anche per le variabili nelle altre funzioni; per esempio, la variabile **i** in **getline** non ha alcuna relazione con la **i** in **copy**. Ogni variabile locale di una routine viene creata solamente quando la funzione viene chiamata e *sparisce* quando la funzione termina. È per questa ragione che alcune variabili sono generalmente conosciute come variabili *automatiche*, seguendo la terminologia di altri linguaggi. D'ora in poi useremo il termine automatiche per indicare queste variabili locali dinamiche (il Capitolo 4 discute l'argomento dell'allocazione **static** in cui le variabili locali mantengono il proprio valore tra le chiamate di funzioni).
Siccome le variabili automatiche vanno e vengono con le chiamate di funzione, non mantengono il proprio valore tra una chiamata e l'altra e bisogna esplicitamente assegnarle ad ogni entrata. Se non vengono assegnate, conterranno dei valori casuali.
Come alternativa alle variabili automatiche, è possibile definire variabili che siano *esterne* a tutte le funzioni, cioè variabili globali a cui ogni funzione che ne ha interesse può accedere per nome. (Questo meccanismo è simile al **COMMON** del Fortran o all'**EXTERNAL** del **PL/1**). Poiché le variabili esterne sono globalmente accessibili, possono essere usate al posto degli argomenti per comunicare dati tra le funzioni. Inoltre, poiché le variabili esterne rimangono permanentemente, invece di apparire e scomparire quando le funzioni vengono chiamate e terminate, conservano il proprio valore anche dopo che sono terminate le funzioni che le hanno assegnate.
Una variabile esterna deve essere *definita* al di fuori di ogni funzione; in questo modo viene allocata la variabile in memoria. Inoltre la variabile deve essere *dichiarata* in ogni funzione che vuole accedervi. Ciò può essere fatto sia con un'esplicita dichiarazione **extern** sia implicitamente nel contesto. Entrando nel concreto, riscriviamo il programma della linea più lunga con **linea, salva** e **max** come variabili esterne. Questo richiede il cambiamento di chiamate, dichiarazioni e corpi di tutte e tre le funzioni.

```
#define    MAXLINEE  1000 /* massima ampiezza della linea in input */

char linea[MAXLINEE]; /* linea in input */
char salva[MAXLINEE]; /* la linea piu' lunga, salvata */
int  max; /* lunghezza massima finora incontrata */

main()  /* trova la linea piu' lunga; versione specializzata */
{
    int lung;
    extern int max;
    extern char salva[];

    max = 0;
    while ((lung = getline()) > 0)
    if (lung > max) {
        max = lung;
        copy();
    }
    if (max > 0)    /* c'era una linea */
        printf("%s", salva);

}

getline() /* versione specializzata */
{
    int c, i;
    extern char linea[];

    for (i = 0; i < MAXLINEE -1
         && (c=getchar()) != EOF && c != '\n'; ++i)
            linea[i] = c;
    if (c == '\n') {
        linea[i] = c;
        ++i;
    }
    linea[i] = '\0';
    return(i);
}
```

```
copy()     /* versione specializzata */
{
     int i;
     extern char linea[], salva[];

     i = 0;
     while ((salva[i] = linea[i]) != '\0')
          ++i;
}
```

Le variabili esterne di **main, getline** e **copy** vengono *definite* dalle prime linee dell'esempio precedente, che specificano i loro tipi e le allocano in memoria. Sintatticamente, le definizioni esterne sono tali e quali alle dichiarazioni usate fin'ora, ma, poiché compaiono al di fuori di ogni funzione, le variabili sono esterne. Prima che una funzione possa usare una variabile esterna, bisogna rendere noto alla funzione il nome della variabile. Un modo per farlo è scrivere una *dichiarazione* **extern** all'interno della funzione; la dichiarazione è uguale a prima eccetto l'aggiunta della parola chiave **extern**.
In certe circostanze, la dichiarazione **extern** può essere omessa: se la dichiarazione esterna di una variabile compare nel programma *prima* del suo uso in una particolare funzione, non c'è bisogno di una dichiarazione **extern** nella funzione. Le dichiarazioni **extern** in **main, getline** e **copy** sono così ridondanti. Infatti, la pratica comune è quella di collocare le definizioni di tutte le variabili esterne all'inizio del file sorgente e di omettere poi tutte le dichiarazioni **extern**.
Se il programma risiede su più file sorgenti, e una variabile è definita ad esempio in *file 1* ed usata in *file 2*, allora c'è bisogno di una dichiarazione **extern** all'interno di *file 2* per connettere le due occorrenze della variabile. Questo argomento viene trattato a fondo nel Capitolo 4.
Si noti che in questo paragrafo stiamo usando attentamente le parole *dichiarazione* e *definizione* quando ci riferiamo alle variabili esterne. "Definizione" si riferisce al contesto in cui la variabile viene creata o le viene assegnata memoria; "dichiarazione" si riferisce ai contesti in cui viene affermata la natura della variabile ma non viene allocata memoria. A questo proposito, c'è la tendenza a porre tutto in vista con le variabili **extern** poiché semplifica le comunicazioni — la lista di argomenti è breve e le variabili sono sempre presenti quando servono. Ma le variabili esterne ci sono sempre anche se non si vogliono. Questo stile nello scrivere programmi è pericoloso poiché conduce a programmi in cui non sono chiari i collegamenti tra i dati — le variabili possono essere cambiate in maniera inaspettata ed anche involontaria ed il programma è difficilmente modificabile al momento necessario. La seconda versione del programma delle linee più lunghe è inferiore alla prima, in parte per queste ragioni ed in parte perché distrugge la generalità di due utili funzioni inserendovi dentro i nomi delle variabili che vengono manipolate.

Esercizio 1-18. Il test nell'istruzione **for** in **getline** è piuttosto grossolano. Riscrivere il programma in maniera più chiara ma conservando lo stesso funzionamento sulla fine del file ed overflow del buffer. Questo funzionamento è il più ragionevole?

1.11 Sommario

A questo punto abbiamo scoperto ciò che potrebbe essere chiamato il nucleo convenzionale del C. Con l'apporto di queste costruzioni basilari, è possibile scrivere programmi utili di considerevole grandezza, e probabilmente sarebbe una buona idea se ci si fermasse abbastanza per farlo. Gli esercizi che seguono intendono dare suggerimenti per programmi più complessi di quelli presentati in questo capitolo.
Con questa parte di C sotto controllo, vale la pena di continuare a leggere, in quanto è nelle caratteristiche illustrate nei prossimi capitoli che diventano evidenti la potenza e l'espressività del linguaggio.

Esercizio 1-19. Scrivere un programma **detab** che rimpiazzi i tab dell'input con il numero appropriato di spazi per spaziare fino al successivo tab. Ipotizzare una tabulazione fissa, diciamo ogni *n* posizioni.

Esercizio 1-20. Scrivere un programma **entab** che rimpiazza stringhe di blank con il minimo numero di tab e bianchi per raggiungere la stessa spaziatura. Usare la stessa lunghezza del tab come per **detab**.

Esercizio 1-21. Scrivere un programma che "spezzi" lunghe linee di input dopo l'ultimo carattere non bianco che compare prima dell'ennesima colonna di input, in cui *n* è un parametro. Ci si assicuri che il programma si comporti intelligentemente con linee molto lunghe, e che non ci siano spazi o tab prima della colonna specificata.

Esercizio 1-22. Scrivere un programma che rimuove tutti i commenti da un programma C. Non ci si dimentichi di considerare in maniera appropriata stringhe tra apici e costanti carattere.

Esercizio 1-23. Scrivere un programma che analizzi in un programma C rudimentali errori sintattici come parentesi non bilanciate, quadre e graffe. Non ci si dimentichi degli apici singoli e doppi e dei commenti. (Questo programma è difficile se lo si considera in tutta la sua interezza).

CAPITOLO 2

TIPI, OPERATORI ED ESPRESSIONI

Variabili e costanti sono i dati che vengono manipolati in un programma. Le dichiarazioni elencano le variabili da usarsi, ne affermano il tipo e in alcuni casi anche i valori iniziali. Gli operatori specificano quali azioni debbano essere compiute su di esse. Le espressioni combinano costanti e variabili in modo da produrre nuovi valori. Questi sono gli argomenti di questo capitolo.

2.1 Nomi di variabili

Ci sono alcune restrizioni riguardo ai nomi delle variabili e delle costanti simboliche. I nomi sono costituiti da lettere e cifre; il primo carattere deve essere una lettera. La sottolineatura "_" è considerata come una lettera; è utile per migliorare la leggibilità di lunghi nomi di variabili. Le forme maiuscola e minuscola sono differenti; è pratica tradizionale del C usare la forma minuscola per i nomi delle variabili e quella maiuscola per le costanti simboliche.
Sono significativi soltanto i primi otto caratteri di un nome interno, anche se possono esserne usati molti altri. Per quanto riguarda i nomi esterni quali i nomi di funzioni e di variabili esterne, il numero può risultare inferiore ad otto, dato che i nomi esterni sono usati da vari assemblatori e loaders. I dettagli sono riportati nell'Appendice A. Inoltre, le parole chiave come **if, else, int, float,** ecc. sono da considerarsi *riservate*: non possono essere usate come nomi di variabili (devono essere in forma minuscola)-
Naturalmente è opportuno scegliere nomi di variabili che abbiano significato, siano in relazione con lo scopo della variabile, e che sia improbabile che generino confusione per via della forma tipografica.

2.2 Tipi e dimensioni dei dati

Ci sono solo pochi tipi fondamentali di dati nel C:

char	un solo byte, in grado di contenere un carattere del set disponibile.
int	un numero intero, che riflette tipicamente la dimensione naturale degli interi sulla macchina ospite.
float	singola precisione in virgola mobile.
double	doppia precisione in virgola mobile

Inoltre, ci sono alcuni qualificatori che possono venire applicati a **int**: **short**, **long** e **unsigned**. **short** e **long** si riferiscono a numeri interi di diversa dimensione. I numeri **unsigned** obbediscono alle leggi dell'aritmetica modulo 2^n, dove *n* è il numero di bit in un **int**; i numeri **unsigned** sono sempre positivi. Le dichiarazioni per quanto riguarda i qualificatori appaiono come:

```
short int x;
long int y;
unsigned int z;
```

In tali casi la parola **int** può essere omessa, come abitualmente succede.
La precisione di questi oggetti dipende dal tipo di macchina a disposizione; la tavola qui sotto mostra alcuni valori rappresentativi.

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	EBCDIC	ASCII
char	8 bits	9 bits	8 bits	8 bits
int	16	36	32	32
short	16	36	16	16
long	32	36	32	32
float	32	36	32	32
double	64	72	64	64

short e **long** dovrebbero fornire differenti lunghezze di numeri interi quando è più pratico; **int** rifletterà la dimensione più appropriata per la macchina specificatamente considerata. Come potete vedere, ogni compilatore è libero di interpretare **short** e **long** nel modo più idoneo al suo hardware. Quello su cui si può tranquillamente contare è che **short** non è più lungo di **long**.

2.3 Le costanti

Le costanti **int** e **float** sono già state trattate, non resta che notare che la normale notazione scientifica

```
123.456e-7
```

oppure

```
0.12E3
```

per i **float** è permessa. Ogni costante in virgola mobile viene presa per **double**, così che la notazione **"e"** serve per entrambi i **float** e **double**.
Le costanti **long** sono scritte nello stile **123L**. Una normale costante intera che sia troppo lunga per rientrare in un **int** è anch'essa considerata come **long**.
Esiste una notazione per le costanti ottali ed esadecimali: uno **0** (zero) in prima posizione in una costante intera implica la notazione ottale; una coppia **0x** o **0X** in prima posizione

indica la notazione esadecimale. Per esempio, il decimale **31** può essere riscritto **037** in ottale e **0x1f** o **0X1F** in esadecimale. Le costanti esadecimali e ottali possono anche essere seguite da **L** per farle diventare **long**.
Una *costante carattere* è un carattere unico scritto tra singoli apici come in **'x'**. Il valore di una costante di questo tipo è il valore numerico del carattere nel set della macchina. Per esempio, nel set **ASCII**, il carattere zero, o '0', è 48, ed in **EBCDIC** è 240, entrambi molto differenti dal valore numerico 0. Scrivere '0' al posto di un valore numerico come 48 o 240, rende il programma indipendente da quel particolare valore. Le costanti carattere partecipano alle operazioni numeriche così come ogni altro numero, anche se vengono usate nella maggior parte dei casi in confronti con altri caratteri. Un paragrafo successivo tratterà le regole di conversione.
Alcuni caratteri non letterali possono essere rappresentati come costanti di tipo carattere per mezzo di sequenze di escape come **\n** (newline), **\t** (tab), **\0** (null), \\ (backslash), \' (singolo apice) ecc.; hanno l'aspetto di due caratteri, ma in realtà ne rappresentano uno solo. Inoltre, si può generare uno schema a bit della dimensione di un byte scrivendo

'\ddd'

dove *ddd* è composto da una fino a tre cifre ottali come in

```
#define    SALTOPAGINA    '\014'    /* salto pagina ASCII */
```

La costante **'\0'** rappresenta il carattere con valore zero. Spesso si scrive **'\0'** al posto di **0** per enfatizzare la natura di tipo carattere di alcune espressioni.
Espressioni costanti sono espressioni che coinvolgono solo costanti. Tali espressioni vengono valutate durante la compilazione invece che in esecuzione, e di conseguenza possono venire impiegate dovunque possa esserlo una costante, come in

```
#define    MAXLINEA    1000
char linea[MAXLINEA+1];
```

oppure

```
secondi = 60 * 60 * ore;
```

Una costante stringa è una sequenza di zero o più caratteri racchiusa tra doppi apici, come in

```
"Sono una stringa"
```

oppure

```
""    /* una stringa nulla */
```

I doppi apici non fanno parte della stringa, ma servono solo per delimitarla. Le stesse sequenze usate per le costanti di tipo carattere si applicano alle stringhe; **\"** rappresenta il carattere doppio apice.

Tecnicamente una stringa è un vettore i cui elementi sono singoli caratteri. Il compilatore colloca automaticamente il carattere \0 alla fine di ciascuna stringa di questo tipo, in modo che i programmi possano agevolmente trovarne la fine. Questa rappresentazione dimostra che non esiste un vero limite alla lunghezza possibile di una stringa, ma è comunque necessario che i programmi la scorrano completamente per determinarne la lunghezza. La memoria fisica richiesta è di una posizione in più rispetto al numero di caratteri scritti tra virgolette. La seguente funzione **strlen(s)** restituisce la lunghezza di una stringa di caratteri **s**, escludendo il **\0** finale.

```
strlen(s)     /* ritorna la lunghezza di s */
char s[];
{
     int i;

     i = 0;
     while (s[i] != '\0')
          ++i;
     return(i);
}
```

Fate attenzione a distinguere tra una costante di tipo carattere e una stringa che contiene un solo carattere: **'x'** non è lo stesso di **"x"**. Il primo è un carattere solo, che serve per produrre il valore numerico della lettera *x* nel set di caratteri della macchina. Il secondo è una stringa di caratteri che contiene un carattere (la lettera x) e un **\0**.

2.4 Le dichiarazioni

Tutte le variabili devono essere dichiarate prima dell'uso, sebbene alcune dichiarazioni possano essere effettuate implicitamente dal contesto. Una dichiarazione specifica un tipo, ed è seguita da una lista di variabili di quel tipo, come in

```
int     minimo, massimo, passo;
char    c, linea[1000];
```

Le variabili possono essere distribuite nell'ambito delle dichiarazioni in qualsiasi modo; le liste sopra riportate potrebbero essere ugualmente scritte come

```
int     minimo;
int     massimo;
int     passo;
char    c;
char    linea[1000];
```

Quest'ultima forma occupa più spazio, ma è consigliabile per poter aggiungere un commento a ciascuna dichiarazione o per successive modifiche.

Le variabili possono anche essere inizializzate al momento della loro dichiarazione, sebbene ci siano alcune restrizioni. Se il nome è seguito da un segno di uguale e da una costante, questa serve come valore iniziale, come in

```
char      backslash = '\\';
int       i = 0;
float     eps = 1.0e-5;
```

Se la variabile in questione è esterna o statica, l'inizializzazione viene effettuata una volta sola, concettualmente prima che il programma sia eseguito. Le variabili automatiche inizializzate esplicitamente sono inizializzate ogni volta che la funzione di cui fanno parte viene chiamata. Le variabili automatiche per cui non è specificato un esplicito valore iniziale hanno valori indefiniti. Le variabili esterne e statiche vengono inizializzate a zero per default, ma è buona norma dichiarare comunque l'inizializzazione.
Discuteremo l'inizializzazione più avanti quando introdurremo nuovi tipi di dati.

2.5 Operatori Aritmetici

Gli operatori aritmetici binari sono +, **-**, *, /, e l'operatore di modulo **%**. Esiste l'operatore unario –, ma non quello +.
La divisione intera tronca ogni parte frazionaria. L'espressione

```
x % y
```

produce il resto della divisione di **x** per **y**, e perciò è zero quando **y** è divisibile esattamente per **x**. Per esempio, un anno è bisestile se è divisibile per 4 ma non per 100 eccetto quegli anni divisibili per 400 che *sóno* bisestili. Perciò

```
if (anno % 4 == 0 && anno % 100 != 0 || anno % 400 == 0)
```

è un anno bisestile

```
else
```

non lo è

L'operatore % non può essere applicato a **float** o a **double**.
Gli operatori + e – hanno la medesima priorità che è inferiore alla (identica) priorità di *, /, % che sono a loro volta inferiori al meno unario. Gli operatori aritmetici si raggruppano da sinistra a destra. (Una tavola alla fine del capitolo riporta in breve la priorità e l'associatività fra tutti gli operatori). L'ordine di valutazione non è specificato per gli operatori associativi e commutativi quali * e +; il compilatore può riorganizzare un calcolo con parentesi che coinvolge uno di questi. Così **a+(b+c)** può essere valutato come se fosse **(a+b)+c**.
Normalmente ciò non porta a differenze significative, ma se è richiesto un ordine particolare, devono essere impiegate esplicite variabili temporanee.
I provvedimenti adottati in caso di overflow o underflow dipendono dalla macchina.

2.6 Operatori Relazionali e Logici

Gli operatori relazionali sono

```
>    >=    <    <=
```

Hanno tutti la stessa priorità. Gli operatori di eguaglianza li seguono immediatamente in termini di priorità:

```
==    !=
```

che hanno la medesima precedenza. I relazionali hanno una priorità inferiore agli operatori aritmetici, in questo modo espressioni del tipo **i** < **lim-1** sono considerate come **i** < **(lim-1)**, così come ci si aspetta.
Maggior interesse presentano gli operatori logici di connessione **&&** e **||**. Le espressioni collegate da **&&** o **||** vengono valutate da sinistra a destra, e la valutazione cessa nel momento in cui viene accertata la verità o falsità del risultato. Queste caratteristiche sono cruciali nello scrivere programmi che funzionino. Per esempio, qui di seguito riportiamo un ciclo tratto dalla funzione di input **getline** di cui parlammo nel Capitolo 1.

```
for (i=0; i<lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
    s[i] = c;
```

Ovviamente, prima di leggere un nuovo carattere, è necessario controllare che ci sia posto per immagazzinarlo nell'array **s**; per cui il controllo **i**<**lim-1** *deve* essere effettuato prima; non solo, ma in caso di controllo negativo non deve proseguire nella lettura di un altro carattere.
Di conseguenza sarebbe un inconveniente se **c** venisse confrontato con **EOF** prima che **getchar** venga chiamata: la chiamata deve verificarsi prima che venga analizzato il carattere contenuto in **c**.
La priorità di **&&** è maggiore di quella di **||** ed entrambe sono inferiori a quella degli operatori relazionali e di eguaglianza, perciò espressioni quali

```
i<lim-1 && (c=getchar()) != '\n' && c != EOF
```

non necessitano di parentesi ulteriori. Ma dato che la priorità di != è superiore all'assegnamento, sono necessarie le parentesi in

```
(c = getchar()) != '\n'
```

per ottenere il risultato voluto.
L'operatore unario di negazione **!** converte un operando non zero o vero in 0, e un operando zero o falso in 1. Un uso comune di **!** si rileva in costruzioni del tipo

```
if (!inparola)
```

invece che

```
if (inparola == 0)
```

È arduo decidere quale sia la forma migliore. Costruzioni del tipo **!inparola** si leggono agevolmente ("se non è in parola"), ma quelle più complesse possono essere difficili da capire.

Esercizio 2-1. Scrivere un ciclo equivalente al **for** prima visto, senza usare &&.

2.7 Conversioni di Tipo

Quando operandi di tipo differente appaiono nelle espressioni, vengono convertiti in un tipo comune conformemente ad un piccolo insieme di regole. In generale le sole conversioni che avvengono automaticamente sono quelle che hanno senso, quali le conversioni di un intero in floating point in un'espressione del tipo **f+i**. Espressioni che non hanno senso, per esempio usare un **float** come indice, non vengono ammesse.
Innanzitutto, i **char** e gli **int** possono essere liberamente mischiati nelle espressioni aritmetiche: ogni **char** nelle espressioni è automaticamente convertito in un **int**. Questo permette una certa flessibilità in taluni casi di trasformazione di caratteri. Una di queste è esemplificata dalla funzione **atoi**, che opera la conversione da una stringa di cifre nel suo valore numerico equivalente.

```
atoi(s)   /* converte s in intero */
char s[];
{
    int i, n;

    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + s[i] - '0';
    return(n);
}
```

Ricordando ciò che abbiamo discusso nel Capitolo 1, l'espressione

```
s[i] - '0'
```

risulta nel valore numerico del carattere immagazzinato in **s[i]** dato che i valori di '0', '1', ecc. formano una sequenza positiva continua crescente.
Un'altro esempio di conversione da **char** a **int** è la funzione **lower** che trasforma un singolo carattere in forma minuscola solo per il *set di caratteri ASCII*. Se il carattere non è maiuscolo, **lower** lo riporta inalterato.

```
lower(c)  /* converte c in minuscolo; solamente ASCII */
int c;
{
    if (c >= 'A' && c <= 'Z')
        return(c + 'a' - 'A');
    else
        return(c)
}
```

Questo funziona per il set **ASCII**, dato che le lettere maiuscole e minuscole corrispondenti si trovano tra loro ad una distanza predeterminata come valori numerici ed entrambi gli alfabeti sono contigui — non ci sono altro che lettere tra A e Z. Quest'ultima osservazione *non* è vera riguardo il set di caratteri **EBCDIC** (**IBM** 360/370), di conseguenza questo codice non funziona su quei sistemi in quanto trasforma altri caratteri oltre alle lettere.
Inoltre esiste un aspetto particolarmente delicato che riguarda la conversione dei caratteri in numeri interi. Il linguaggio non specifica se le variabili di tipo **char** sono quantità con segno oppure no. Qualora un **char** sia convertito in un **int**, potrà in qualche caso produrre un numero intero *negativo*? Sfortunatamente ciò varia da macchina a macchina e riflette le differenze nell'architettura. Su alcune macchine (PDP-11 per esempio), un **char** il cui bit posto più a sinistra è 1, verrà trasformato in un numero intero negativo (estensione di segno). Su altre un **char** diventa **int** aggiungendo degli zeri all'estremità sinistra, risultando perciò sempre positivo.
Nella stessa definizione del C si garantisce che ogni carattere del set standard della macchina non sarà mai negativo, cosicché questi caratteri possono liberamente venire usati nelle espressioni come quantità positive. Ma insiemi arbitrari di bit immagazzinati nelle variabili carattere possono risultare negativi su alcune macchine, mentre risultano positivi su altre.
La più comune ricorrenza di questa situazione si verifica quando il valore -1 è usato per **EOF**. Si consideri il codice

```
char c;

c = getchar();
if (c == EOF)
    ...
```

Su una macchina che non effettua l'estensione di segno, **c** è sempre positivo perché si tratta di un **char**; eppure **EOF** è negativo. Come risultato, il test fallisce sempre. Per evitare ciò, siamo stati attenti ad usare **int** invece di **char** per ogni variabile che contiene un valore di ritorno di **getchar**.
La vera ragione per usare **int** al posto di **char** non ha a che fare con possibili problemi di estensione di segno. La ragione è che **getchar** deve restituire tutti i possibili caratteri (così che può venire usata per la lettura di un qualsiasi input) e inoltre un valore distinto **EOF**. Perciò il suo valore *non può* essere rappresentato come un **char**, ma deve invece essere memorizzato come un **int**.
Un'altra utile forma di conversione automatica di tipo è che le espressioni relazionali quali **i** > **j** e le espressioni logiche collegate da **&&** e **||** vengono definite in modo da avere valore 1 se sono vere e 0 se sono false. Perciò l'assegnamento

```
isdigit = c >= '0' && c <= '9';
```

pone **isdigit** a 1 se **c** è una cifra, e se non lo è lo pone a 0. (Nei predicati **if, while, for**, ecc., "vero" significa solo "non zero").
Le conversioni aritmetiche implicite si comportano per lo più così come ci si aspetta. In generale se un operatore come + o * che si applica a due operandi (un "operatore binario") ha operandi di tipo differente, il tipo "inferiore" viene promosso a "superiore"

prima che l'operazione prosegua. Il risultato è del tutto superiore. Più precisamente per ogni operatore aritmetico si applica la seguente sequenza di regole di conversione.

> **char** e **short** sono convertiti in **int** e **float** è trasformato in **double**.
>
> Poi se uno dei due operandi è **double**, l'altro è convertito in **double**, e il risultato è **double**.
>
> Altrimenti se uno dei due operandi è **long**, l'altro è convertito in **long**, e il risultato è **long**.
>
> Altrimenti se uno dei due operandi è **unsigned**, l'altro è convertito in **unsigned**, e il risultato è **unsigned**.
>
> Altrimenti gli operandi devono essere **int** e il risultato **int**.

Notare che tutti i **float** in una espressione sono convertiti in **double**; tutta l'aritmetica in floating point in C viene risolta in doppia precisione.
Le conversioni avvengono attraverso l'assegnamento; il valore della parte destra è convertito nel tipo di sinistra, che è quello del risultato. Un carattere è trasformato in un numero intero, sia per mezzo dell'estensione di segno sia senza, così come descritto precedentemente. L'operazione inversa, da **int** a **char**, è eseguita correttamente — i bit più significativi in eccesso vengono semplicemente tralasciati. Perciò in

```
int i;
char c;

i = c;
c = i;
```

il valore di **c** è immutato. Questo è vero sia che venga effettuata o no l'estensione di segno.
Se **x**è **float** ed **i** è **int**, allora

```
x = i
```

e

```
i = x
```

entrambe determinano conversioni; quella da **float** a **int** provoca troncamento di ogni parte decimale. **double** è trasformato in **float** per arrotondamento. Gli **int** più lunghi vengono convertiti in quelli più corti o in **char** per eliminazione dei bit più significativi.
Poiché un argomento di funzione è un'espressione, le conversioni di tipo avvengono anche quando gli argomenti sono passati alle funzioni: in particolare **char** e **short** diventano **int**, e **float** diventa **double**. Ciò spiega il perché abbiamo considerato gli argomenti di funzione come **int** e **double** perfino quando la funzione viene chiamata con **char** e **float**.

Infine, conversioni esplicite di tipo possono essere forzate in una qualsiasi espressione con un costrutto chiamato *cast*. Nel costrutto

(nome-di-tipo) espressione

l'espressione è trasformata nel tipo indicato per mezzo delle regole di conversione sopra riportate. Il preciso significato di un cast è infatti come se la *espressione* fosse assegnata ad una variabile del tipo specificato, che è quindi usato al posto dell'intera costruzione. Per esempio la routine di libreria **sqrt** riceve un argomento **double**, e non produrrà niente di sensato se inavvertitamente gli viene passato qualcosa di diverso. Così se **n** è un numero intero,

```
sqrt((double) n)
```

trasforma **n** in **double** prima di passarlo a **sqrt**. (Notare che il cast produce il *valore* **n** nel tipo appropriato; il contenuto specifico di **n** non muta). L'operatore cast ha la stessa priorità degli altri operatori unari, così come viene sintetizzato nella tavola alla fine di questo capitolo.

Esercizio 2-2. Scrivere la funzione **htoi(s)**, che converte una stringa di cifre esadecimali nel suo valore intero equivalente. Le cifre consentite sono da **0** a **9**, da **a** ad **f** e da **A** a **F.**

2.8 Operatori di Incremento e Decremento

Il C fornisce due operatori inusuali per incrementare e decrementare le variabili. L'operatore di incremento ++ aggiunge 1 al suo operando; l'operatore di decremento -- toglie 1. Abbiamo usato frequentemente ++ per incrementare le variabili, come in

```
if (c == '\n')
    ++nl;
```

L'aspetto particolare è che ++ e -- possono essere usati sia come operatori prefissi (prima della variabile, come in ++**n**) che come postfissi (dopo la variabile: **n**++). In entrambi i casi l'effetto che si ottiene è l'incremento di **n**. Ma l'espressione ++**n** incrementa **n** *prima* di usarne il valore, mentre **n**++ aumenta **n** *dopo* che ne è stato usato il valore. Questo significa che in un contesto in cui si stia usando il valore, non solo l'effetto prodotto, ++**n** e **n**++ sono differenti. Se **n** è 5, allora

```
x = n++;
```

assegna 5 ad **x**, ma

```
x = ++n;
```

assegna 6 ad **x**. In entrambi i casi, **n** diventa 6. Gli operatori di incremento e decremento possono essere applicati solo alle variabili; un'espressione di tipo **x=(i+j)++** non è consentita.
In un contesto in cui non si richiede alcun valore, ma l'effetto puro e semplice di incremento come in

```
if (c == '\n')
    nl++;
```

si sceglie il prefisso o il postfisso a seconda dei gusti. Ma esistono situazioni in cui sono esplicitamente richiesti l'uno o l'altro. Infatti, considerate la funzione **squeeze(s, c)** che toglie tutte le possibili ricorrenze del carattere **c** nella stringa **s**.

```
squeeze(s, c)    /* rimuove tutti i c da s */
char s[];
int c;
{
    int i, j;

    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

Ogni volta che ricorre un carattere non-**c**, viene copiato nella posizione corrente **j** e solo a questo punto **j** viene incrementato in modo da essere pronto per il prossimo carattere. Ciò è esattamente equivalente a

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

Un'altro esempio di una costruzione simile è quello della funzione **getline** che scrivemmo nel Capitolo 1, in cui possiamo sostituire

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

con il più compatto

```
if (c == '\n')
    s[i++] = c;
```

Come terzo esempio, la funzione **strcat(s, t)**concatena la stringa **t** alla fine della stringa **s**. strcat presuppone che in **s** ci sia abbastanza spazio per contenere la combinazione.

```
strcat(s, t)     /* collega t alla fine di s */
char s[], t[];   /* s dev'essere abbastanza grande */
{
     int i, j;

     i = j = 0;
     while (s[i] != '\0')    /* fine di s trovata */
          i++;
     while ((s[i++] = t[j++]) != '\0')  /* copia t */
          ;
}
```

Mentre ogni carattere viene copiato da **t** ad **s**, il suffisso ++ viene applicato ad entrambi **i** e **j** per fare in modo che i caratteri siano in posizione per il successivo passaggio attraverso il ciclo.

Esercizio 2-3. Scrivere un'altra versione di **squeeze(s1, s2)** che rimuova ogni carattere in **s1** uguale ad uno qualsiasi dei caratteri della *stringa* **s2**.

Esercizio 2-4. Scrivere la funzione **any(s1, s2)** che riporta la prima posizione nella stringa **s1** dove ricorra uno qualsiasi dei caratteri della stringa **s2**, o -1 se s1 non contiene caratteri di **s2**.

2.9 Operatori Logici orientati al Bit

Il C è dotato di un certo numero di operatori per la manipolazione dei bit; questi operatori non devono essere applicati a **float** o a **double**.

```
&    AND a bit
|    OR inclusivo a bit
^    OR esclusivo a bit
<<   shift a sinistra
>>   shift a destra
~    complemento a uno (unario)
```

L'operatore di **AND** sui bit **&** viene spesso impiegato per mascherare un insieme di bit; per esempio

```
c = n & 0177
```

riduce tutto a zero eccetto i 7 bit meno significativi di **n**. L'operatore di **OR** sui bit **|** viene impiegato per "accendere" i bit.

```
x = x | MASK
```

pone ad 1 in **x** i bit che sono a 1 in **MASK**.
Si dovrebbero distinguere chiaramente gli operatori sui bit **&** e **|** dai connettivi logici **&&** e **||**, che implicano una valutazione da sinistra a destra di un valore di verità. Per esempio, se **x** è 1 e **y** 2, allora **x** & **y** è zero mentre **x && y** è uno. (Perché?)

Gli operatori di shift << e >> operano shift a sinistra e a destra del loro operando sinistro, di un numero di posizioni dato dall'operando destro. Perciò **x << 2** fa slittare a sinistra **x** di due posizioni, riempiendo i bit lasciati liberi con 0. Questo è equivalente alla moltiplicazione per 4. Lo shift a destra di una quantità unsigned riempie i bit liberi con 0. Lo shift a destra di una quantità con segno riempirà con bit di segno ("shift aritmetico") su alcune macchine come il **PDP-11** e con bit a 0 ("shift logico") su altre. L'operatore unario ~ produce il complemento a uno di un intero; cioè, trasforma ogni bit 1 in 0 e viceversa. Questo operatore trova tipicamente impiego in espressioni come

```
x & ~077
```

che maschera gli ultimi 6 bit di **x** a zero. Si noti che **x &~077** è indipendente dalla lunghezza della parola ed è perciò preferibile a, per esempio, **x & 0177700**, che presuppone che **x** sia una quantità in 16 bit. La forma portabile non comporta altro costo dato che **~077** è un'espressione costante e perciò valutata in compilazione.
Per illustrare l'uso di alcuni degli operatori sui bit, si consideri la funzione **getbits(x, p, n)** che restituisce (allineato a destra) il campo di **n** bit di **x** che inizia alla posizione **p**. Supponiamo che la posizione del bit 0 sia all'estremità destra e che **n** e **p** siano valori positivi sensati. Per esempio, **getbits(x, 4, 3)** riporta i tre bit in posizione 4, 3 e 2 allineati a destra.

```
getbits(x, n, p)  /* prende n bit dalla posizione p */
unsigned x, p, n;
{
    return((x >> (p+1-n)) & ~(~0 << n));
}
```

x >> (p+1-n) sposta il campo desiderato all'estremità destra della parola. Dichiarando l'argomento **x** essere **unsigned** assicura che quando viene ottenuto uno shift a destra, i bit liberi saranno rimpiazzati con zero e non con bit con segno, senza tener conto della macchina in cui viene eseguito il programma. **~0** è con tutti i bit uguali a 1; eseguendo uno shift a sinistra di **n** posizioni di bit con **~0 << n** crea una maschera di zeri negli n bit più a destra e di uni nelle altre parti; il complemento di ciò con ~ produce una maschera di uni negli **n** bit più a destra.

Esercizio 2-5. Modificare **getbits** per numerare i bit da sinistra a destra.

Esercizio 2-6. Scrivere una funzione **wordlength()** che calcoli la lunghezza della parola della macchina ospite, cioè il numero di bit in un **int**. La funzione dovrebbe essere portabile nel senso che il medesimo codice sorgente funziona su tutte le macchine.

Esercizio 2-7. Scrivere la funzione **rightrot(n, b)** che ruota a destra il numero intero **n** di **b** bit posizioni.

Esercizio 2-8. Scrivere la funzione **invert(x, p, n)** che inverte (ad es. cambia 1 in 0 e viceversa) gli **n** bit di **x** che iniziano alla posizione **p**, lasciando inalterati gli altri.

2.10 Operatori ed Espressioni di Assegnamento

Espressioni come

```
i = i + 2
```

nelle quali la parte sinistra è ripetuta a destra, possono essere scritte nella forma sintetica

```
i += 2
```

usando un *operatore di assegnamento* come +=.
La maggior parte degli operatori binari (operatori come + che hanno operandi a destra e a sinistra) hanno un operatore di assegnamento *op*= corrispondente, in cui *op* è uno tra i simboli seguenti

```
+   -   *   /   %   <<   >>   &   ^   |
```

Se *e1* e *e2* sono espressioni, allora

e1 op= e2

è equivalente a

e1 = (e1) op (e2)

eccetto che *e1* è calcolato solo una volta. È da notare l'uso delle parentesi intorno a *e2*:

```
x *= y + 1
```

è di fatto

```
x = x * (y + 1)
```

invece che

```
x = x * y + 1
```

Come esempio, la funzione **bitcount** conta il numero di bit a 1 nel proprio argomento intero.

```
bitcount(n)   /* conta i bit a 1 in n */
unsigned n;
{
    int b;

    for (b = 0; n != 0; n >>= 1)
        if (n & 01)
            b++;
    return(b);
}
```

A parte la sinteticità, gli operatori di assegnamento hanno il vantaggio di corrispondere meglio al modo di pensare della gente. Si dice "aggiungere 2 ad **i**" o "incrementare **i** di 2", non "prendere **i**, aggiungergli 2 e rimettere il risultato in **i**". Quindi **i** += **2**. Per di più, nel caso di un'espressione complicata come

```
yyval[yypv[p3+p4] + yypv[p1+p2]] += 2
```

l'operatore di assegnamento fa si che il codice sia più semplice da capire, dato che il lettore non è costretto a controllare che due lunghe espressioni siano in realtà le stesse, o capire perché non lo siano. Un operatore di assegnamento può anche aiutare il compilatore a produrre un codice più efficiente.
Abbiamo già avuto modo di riscontrare il fatto che l'istruzione di assegnamento ha un valore e che può ricorrere in espressioni; l'esempio più comune è

```
while ((c = getchar()) != EOF)
    ...
```

Gli assegnamenti che usano altri operatori di assegnamento (+=, -=, ecc.) possono comparire nelle espressioni, anche se è un caso meno frequente.
Il tipo di un'espressione d'assegnamento è quello del suo operando di sinistra.

Esercizio 2-9. In un sistema numerico in complemento a 2, **x & (x-1)** cancella il bit 1 dell'estremità destra di **x**. (Perché?). Tenendo presente ciò, scrivere una versione più veloce di **bitcount**.

2.11 Espressioni Condizionali

Le istruzioni

```
if (a > b)
    z = a;
else
    z = b;
```

naturalmente calcolano in **z** il massimo tra **a** e **b**. L'*espressione condizionale*, scritta con l'operatore ternario "**?:**", fornisce un modo alternativo di scrivere questa e simili costruzioni. Nell'espressione

e1 ? e2 : e3

l'espressione *e1* viene valutata per prima. Se è non-zero (vera) viene allora valutata l'espressione *e2* e questo risulta essere il valore dell'espressione condizionale. Altrimenti è *e3* ad essere valutata e a rappresentare il valore. Viene valutata soltanto una sola tra *e2* e *e3*. Perciò volendo porre **z** al massimo di **a** e **b**,

```
z = (a > b) ? a : b;    /* z = max(a, b)  */
```

Si noti che l'espressione condizionale è propriamente un'espressione e che può essere perciò usata come qualsiasi altra. Se *e2* e *e3* sono di differenti tipi, il tipo del risultato è determinato dalle regole di conversione discusse precedentemente in questo capitolo. Per esempio, se **f** è un **float**, e **n** è un **int**, allora l'espressione

```
(n > 0) ? f : n
```

è di tipo **double** senza tener conto se **n** sia positivo o negativo.
Le parentesi non sono necessarie intorno alla prima espressione di un'espressione condizionale, dato che la priorità di **?:** è molto bassa, appena al di sopra di quella dell'assegnamento. Sono tuttavia consigliabili, dato che evidenziano meglio la parte condizionale dell'espressione.
L'espressione condizionale spesso porta ad un codice succinto. Ad esempio il ciclo qui riportato stampa **N** elementi di un array, 10 per linea, con ogni colonna separata da uno spazio bianco e con un newline al termine di ogni linea (l'ultima inclusa).

```
for (i = 0; i < N; i++)
    printf("%6d%c", a[i], (i%10 == 9 || i==N-1) ? '\n' : ' ');
```

Viene stampato un newline ogni dieci elementi e dopo l'**N**-esimo. Tutti gli altri elementi sono seguiti da uno spazio. Anche se può apparire complicato, è istruttivo cercare di scriverlo senza l'espressione condizionale.

Esercizio 2-10. Riscrivere la funzione **lower**, che converte lettere maiuscole in minuscole, per mezzo di un'espressione condizionale al posto di **if-else**.

2.12 Priorità e Ordine di valutazione

La tavola qui riportata riassume le regole per la priorità e l'associatività di tutti gli operatori, inclusi quelli non ancora trattati. Gli operatori sulla medesima linea hanno la stessa priorità; le righe sono in ordine di priorità decrescente, così ad esempio ***,/,** e **%** hanno tutti la stessa priorità, che è maggiore di quella di + e −.

Operatore	Associatività
() [] –>	da sinistra a destra
! ~ ++ – – (tipo) * & sizeof	da destra a sinistra
* / %	da sinistra a destra
+ –	da sinistra a destra
< < > >	da sinistra a destra
< <= > >=	da sinistra a destra
== !=	da sinistra a destra
&	da sinistra a destra
^	da sinistra a destra
|	da sinistra a destra
&&	da sinistra a destra
||	da sinistra a destra
? :	da destra a sinistra
= += –= ecc.	da destra a sinistra
, (Capitolo 3)	da sinistra a destra)

Gli operatori –>e . sono usati per accedere ai membri delle strutture; verranno trattati nel Capitolo 6 insieme a **sizeof** (ampiezza di un oggetto). Il Capitolo 5 tratta * (indirezione) e **&** (indirizzo di).
Si noti che la priorità degli operatori logici sui bit **&**,^ e **|** cade sotto == e **!**=. Ciò implica che espressioni di test sui bit quali

```
if ((x & MASK) == 0) ...
```

devono essere dotate di parentesi per ottenere risultati corretti.
Come ricordato in precedenza, le espressioni che riguardano gli operatori associativi e commutativi (*, +, **&**, ^,|) possono essere ristrutturate anche quando sono tra parentesi. Nella maggior parte dei casi ciò non determina comunque alcuna differenza; nelle situazioni dove ciò accade, possono venire impiegate variabili temporanee esplicite per forzare un particolare ordine di valutazione.
C come la maggior parte dei linguaggi non specifica in quale ordine vengono valutati gli operandi di un operatore. Per esempio, in un'istruzione del tipo

```
x = f() + g();
```

f può essere valutata prima di **g** o viceversa; perciò se **f** o **g** alterano una variabile esterna da cui dipende una delle due, **x** può dipendere dall'ordine di valutazione. Ancora una volta possono essere immagazzinati risultati intermedi in variabili temporanee per assicurare una particolare sequenza.
Similmente, non è specificato l'ordine in cui vengono valutati gli argomenti delle funzioni, così l'istruzione

```
printf("%d %d\n", ++n, potenza(2, n));    /* ERRATO */
```

può produrre risultati (e così fa) diversi su macchine diverse, secondo che **n** venga incrementato o meno prima che **potenza** venga chiamata. La soluzione è certamente

```
++n;
printf("%d %d\n", n, potenza(2, n));
```

Le chiamate di funzione, le istruzioni di assegnamento nidificate e gli operatori di incremento e decremento causano "effetti collaterali" — qualche variabile viene alterata come un sottoprodotto della valutazione di un'espressione. Nelle espressioni che implicano effetti collaterali, possono instaurarsi sottili dipendenze dell'ordine di memorizzazione delle variabili che prendono parte all'espressione. Una tipica situazione poco felice è esemplificata nell'istruzione

```
a[i] = i++;
```

Il problema è se l'indice è il vecchio o il nuovo valore di **i**. Il compilatore può ottenere questo in molti modi, e produrre risposte differenti a seconda dell'interpretazione che ne da. Quando si verificano effetti collaterali il da farsi è lasciato alla discrezione del compilatore, dato che l'ordine migliore dipende fortemente dall'architettura della macchina.
Il senso di questa discussione è che scrivere un codice che dipende da un ordine di valutazione è una cattiva pratica di programmazione in ogni linguaggio. Naturalmente, è necessario sapere quali sono le cose da evitare ma non conoscendo *come* vengono realizzate su differenti macchine, quella piccola lacuna può aiutare nell'evitare errore. (Il verificatore C *lint* sarà utile nel riscontrare la maggior parte delle dipendenze sull'ordine di valutazione).

CAPITOLO 3

STRUTTURE DI CONTROLLO

Le istruzioni di controllo del flusso di un linguaggio specificano l'ordine con cui vengono effettuate le operazioni. Abbiamo appena incontrato negli esempi le più comuni strutture di controllo; ora introdurremo le rimanenti e saremo più precisi circa quelle già viste.

3.1 Istruzioni e Blocchi

Un'*espressione* come **x** = **0**, oppure **i**++, oppure **printf (...)** diventa un'*istruzione* quando è seguita da un punto e virgola come in

```
x = 0;
i++;
printf(...);
```

Nel C il punto e virgola è un terminatore di istruzioni invece che un separatore come nei linguaggi tipo Algol.
Le graffe { e } vengono usate per racchiudere dichiarazioni e istruzioni che formano un'*istruzione composta o blocco* in maniera tale che sono sintatticamente equivalenti ad una singola istruzione. Le graffe che racchiudono le istruzioni di una funzione ne sono un chiaro esempio; un altro è rappresentato dalle graffe che racchiudono più istruzioni dopo un **if, else, while** o **for**. (Ad ogni modo le variabili possono essere dichiarate dentro *qualsiasi* blocco; di ciò parleremo nel Capitolo 4). Non compare mai un punto e virgola dopo la graffa destra che termina un blocco.

3.2 If — Else

L'istruzione **if-else** viene usata per prendere decisioni. Formalmente, la sintassi è

if (*espressione*)
    istruzione–1
else
    istruzione–2

in cui la parte **else** è opzionale.

La *espressione* viene valutata; se è "vera", (se l'*espressione* ha un valore diverso da zero) viene eseguita *istruzione-1*. Se è "falsa", (*espressione* vale zero) e se esiste una parte **else**, al suo posto viene eseguita *istruzione-2*.
Poiché un **if** verifica semplicemente il valore numerico di un'espressione, sono possibili alcune abbreviazioni di codice. La più ovvia è scrivere

if (*espressione*)

invece di

if (*espressione* | = 0)

Talvolta ciò è chiaro e naturale; altre volte criptico.
Siccome la parte **else** di un **if-else** è opzionale, si crea un'ambiguità quando viene omesso un **else** da una sequenza di **if** nidificati. Ciò viene risolto nella usuale maniera — l'**else** viene associato col più recente **else** privo di **if**. Per esempio, in

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

else si riferisce all'**if** più interno, come mostra l'indentazione. Se non è quello che si vuole si devono usare le graffe per forzare l'appropriata associazione:

```
if (n > 0) {
    if (a > b)
        z = a;
}
else
    z = b;
```

L'ambiguità è particolarmente dannosa in situazioni del tipo:

```
if (n > 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("...");
            return(i);
        }
else    /* SBAGLIATO */
    printf("errore - n e' zero\n");
```

L'indentazione mostra chiaramente ciò che si vuole, ma il compilatore non riceve il messaggio e associa l'**else** con l'**if** più interno. Può essere molto difficile scovare un errore di questo tipo.
Ad ogni modo si osservi che c'è un punto e virgola dopo **z** = **a** in

```
if (a > b)
    z = a;
else
    z = b;
```

La ragione è che grammaticalmente, **if** è seguito da un'*istruzione*, e un'espressione-istruzione come **z** = **a** è sempre terminata da punto e virgola.

3.3 Else — If

La costruzione

if (*espressione*)
    istruzione
else if (*espressione*)
    istruzione
else if (*espressione*)
    istruzione
else
    istruzione

si presenta così spesso che richiede una discussione a parte. Questa sequenza di **if** è la maniera più usata per trattare decisioni a più vie. Le *espressioni* vengono valutate in ordine; se una qualsiasi *espressione* è vera, viene eseguita l'*istruzione* associata ad essa e ciò pone termine all'intera catena. Ogni *istruzione* può essere una singola istruzione od un gruppo tra graffe.
L'ultima parte **else** gestisce il caso "nessuno dei precedenti" o "di default" in cui non è stata soddisfatta nessun'altra condizione. A volte non esiste un'azione esplicita di default; in questo caso il costrutto

else
    istruzione

può essere omesso o può essere usato come ricerca di errore per scoprire una condizione "impossibile".
Per illustrare una decisione a tre vie, vediamo una funzione di ricerca binaria che decide se un certo valore **x** compare nell'array ordinato **v**. Gli elementi di **v** devono essere in ordine crescente. La funzione ritorna la posizione (un numero tra **0** e **n−1**) se **x** compare in **v**, altrimenti **−1**.

```
binaria (x, v, n)  /* cerca x in v[0] ... v[n-1] */
int x, v[], n;
{
    int basso, alto, mezzo;

    basso = 0;
    alto = n - 1;
    while (basso <= alto) {
        mezzo = (basso+alto) / 2;
        if (x < v[mezzo])
            alto = mezzo - 1;
        else if (x > v[mezzo])
            basso = mezzo + 1;
        else  /* trovato */
            return(mezzo);
    }
    return(-1);
}
```

La decisione fondamentale è se **x** è minore, maggiore o uguale all'elemento mediano **v[mezzo]** ad ogni ciclo; questo è il classico caso in cui si usa **else-if**.

3.4 Switch

L'istruzione **switch** permette la realizzazione di decisioni multiple e verifica se un'espressione collima con uno tra una serie di valori *costanti*, e conformemente procede nella giusta direzione. Nel Capitolo 1 abbiamo scritto un programma per poter contare le ricorrenze di ogni cifra, spazio bianco e tutti gli altri caratteri usando una sequenza di **if ... else if ... else**. Questo è lo stesso programma con uno **switch**.

```
main()   /* conta cifre, spazi bianchi, altri */
{
    int c, i, nbianchi, naltri, ncifre[10];

    nbianchi = naltri = 0;
    for (i = 0; i < 10; i++)
        ncifre[i] = 0;

    while ((c = getchar()) != EOF)
        switch (c) {
        case '0':
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
            ncifre[c-'0']++;
            break;
        case ' ':
        case '\n':
        case '\t':
            nbianchi++;
            break;
        default:
            naltri++;
            break;
        }

    printf("cifre =");
    for (i = 0; i < 10; i++)
        printf(" %d", ncifre[i]);
    printf("\nspazi bianchi = %d, altri = %d\n",
        nbianchi, naltri);
}
```

Lo **switch** valuta l'espressione intera tra parentesi (in questo programma il carattere **c**) e confronta il suo valore con tutti i casi. Ogni caso dev'essere definito da una costante intera o carattere o da un'espressione costante. Se un caso collima con il valore dell'espressione, l'esecuzione inizia in quel punto. Il caso chiamato **default** viene eseguito se non è stato soddisfatto nessun altro caso. **default** è opzionale; se non compare e se non viene soddisfatto nessuno dei casi, non viene eseguita alcuna azione. Case e default possono apparire in qualunque ordine. I case devono essere tutti differenti.

L'istruzione **break** causa un'uscita immediata dello **switch**. Poiché i case hanno l'unica funzione di etichette, dopo aver superato il codice di un case l'esecuzione *passa* al prossimo, a meno che non si effettuino specifiche azioni per uscire. **break** e **return** sono le vie più comuni per lasciare uno **switch**. L'istruzione **break** può anche essere usata per forzare un'immediata uscita dai cicli **while, for** e **do**, come verrà discusso in questo capitolo.
Il passaggio tra i case ha i suoi pro e contro. Dal lato positivo permette più di un case per una singola azione, come blank, tab o newline in questo esempio, ma implica anche che normalmente ogni caso dev'essere terminato con un **break** per impedire il passaggio al successivo. Il passaggio da un case all'altro non è consigliabile perché porta facilmente a malfunzionamenti quando il programma viene modificato. Con l'eccezione di più label per ogni calcolo, i passaggi dovrebbero essere usati con parsimonia.
È buona abitudine inserire un **break** dopo l'ultimo caso (qui **default**), anche se non è logicamente necessario. Quando in futuro verrà aggiunto alla fine un altro caso, questo accorgimento vi potrà essere molto utile.

Esercizio 3-1. Scrivere la funzione **expand (s, t)** che converte i caratteri come newline e tab in sequenze visibili come **\n** e **\t** mentre copia la strinfa **s** in **t**. Usare uno **switch**.

3.5 Loops – While e For

Abbiamo già incontrato i cicli **while** e **for**. In

```
while (espressione)
    istruzione
```

l'*espressione* viene valutata. Se è diversa da zero, l'*istruzione* viene eseguita e l'*espressione* viene rivalutata. Questo ciclo continua fino a che *espressione* diventa zero, e a quel punto l'esecuzione riprende dopo *istruzione*.
L'istruzione **for**

```
for (expr1; expr2; expr3)
    istruzione
```

è equivalente a

```
exprl;
while (expr2) {
    istruzione
    expr3;
}
```

Grammaticalmente, le tre componenti di un **for** sono espressioni. Generalmente *expr1* ed *expr3* sono assegnamenti o chiamate di funzioni ed *expr2* è un'espressione relazionale.

Ognuna delle tre parti può essere omessa, benché il punto e virgola debba rimanere. Se viene omessa *expr1* o *expr3*, viene semplicemente tolta dall'espansione. Se il test, *expr2*, non è presente, si assume come permanentemente true, così

```
for (;;) {
    ...
}
```

è un ciclo infinito, da interrompere presumibilmente con altri mezzi (come **break** o **return**).
La scelta di usare **while** o **for** è solo una questione di gusto. Per esempio, in

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
    ;   /* salta i caratteri di spaziatura */
```

non c'è inizializzazione o re-inizializzazione, così while sembra più adeguato. Il **for** è chiaramente superiore quando c'è una semplice inizializzazione e re-ininzializzazione, poiché mantiene le istruzioni di controllo del ciclo compatte e visibili all'inizio dello stesso. Ciò è molto ovvio in

```
for (i = 0; i < N; i++)
```

che è il modo del C per scorrere i primi **N** elementi di un array, analogamente ai cicli DO del Fortran o PL/1. L'analogia non è perfetta poiché i limiti di un ciclo **for** possono essere alterati dall'interno del ciclo stesso, e la variabile di controllo **i** mantiene il proprio valore quando per qualsiasi ragione il ciclo termina. Poiché i componenti di un **for** sono espressioni arbitrarie, i cicli **for** non sono limitati a progressioni aritmetiche. Ciononostante non è buono stile forzare calcoli senza legame all'interno di un **for**; è meglio riservarlo per operazioni di controllo del ciclo.
Come esempio più esteso, ecco un'altra versione di **atoi** per convertire una stringa nell'equivalente numerico. Questa versione è più generale; considera spazi bianchi opzionali in testa ed il segno opzionale + o –. (Il Capitolo 4 mostra **atof**, che effettua la medesima conversione per numeri in floating point).
La struttura basilare del programma riflette la forma dell'input:

salta spazi bianchi, se ce ne sono
prendi il segno, se c'è
prendi la parte intera, convertila

Ogni passo fa la propria parte e lascia le cose in maniera pulita per la successiva. L'intero processo termina al primo carattere che non può essere parte di un numero.

```
atoi(s)         /* converte s in intero  */
char s[];
{
    int i, n, segno;

    for (i=0; s[i]==' ' || s[i]=='\n' || s[i]=='\t'; i++)
        ;       /* salta gli spazi bianchi */
    segno = 1;
    if (s[i] == '+' || s[i] == '-')     /* segno */
        segno = (s[i++]=='+') ? 1 : -1;
    for (n = 0; s[i] >= '0' && s[i] <= '9'; i++)
        n = 10 * n + s[i] - '0';
    return(segno * n);
}
```

I vantaggi di mantenere centralizzato il controllo del ciclo sono più evidenti quando ci sono parecchi cicli nidificati. La successiva funzione è uno Shell sort per ordinare array di interi. L'idea basilare di uno Shell sort è che al primo livello vengono confrontati elementi lontani tra di loro, invece che adiacenti come nei semplici ordinamenti di scambio. Ciò tende ad eliminare velocemente gran parte del disordine in maniera tale che i successivi livelli abbiano poco lavoro. L'intervallo tra gli elementi di confronto viene gradualmente decrementato ad uno; a quel punto il sort diventa effettivamente un metodo di interscambio adiacente.

```
shell(v, n)     /* ordina v[0]...v[n-1] in ordine crescente */
int v[], n;
{
    int gap, i, j, temp;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap)  {
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] = temp;
            }
}
```

Ci sono tre cicli nidificati. Il ciclo più esterno controlla il divario tra gli elementi da confrontarsi riducendolo rispetto al valore iniziale **n/2** di un fattore 2 ad ogni passo, fino a che diventa zero. Il ciclo mediano confronta ogni coppia di elementi che vengono separati da **gap**; il ciclo più esterno inverte gli elementi che non sono in ordine. Poiché **gap** verrà infine ridotto ad uno, tutti gli elementi saranno in ogni caso ordinati correttamente. Da notare che la generalità del **for** fa in modo che il ciclo più esterno assuma la stessa forma degli altri anche se non è una progressione aritmetica.
Un ultimo operatore C è la virgola **","**, il cui uso maggiore si trova nell'istruzione **for**. Una coppia di espressioni separate da virgola viene valutata da sinistra a destra, ed il tipo e valore del risultato sono il tipo ed il valore dell'operando destro. Allora in un'istruzione **for** è possibile collocare espressioni multiple nelle varie parti, per esempio per gestire due indici in parallelo. Ciò è illustrato nella funzione **reverse (s)**, che inverte sul posto la stringa **s**.

```
reverse(s)     /* inverte la stringa s sul posto  */
char s[];
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--)  {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

Le virgole che separano gli argomenti nelle funzioni, le variabili nelle dichiarazioni, ecc..., *non* sono operatori virgola, e *non* garantiscono la valutazione da sinistra a destra.

Esercizio 3-2. Scrivere una funzione **expand(s1, s2)** che espande abbreviazioni del tipo **a-z** nella stringa s1 nell'equivalente lista completa **abc...xyz** in **s2**. La funzione implementa lettere di tutti i tipi e cifre e dev'essere preparata ad affrontare casi come **a-b-c** e **a-z0-9** e **-a-z**. (Un'utile convenzione è che un '–' in testa o in coda viene interpretato come una lettera.

3.6 Loops — Do-while

I cicli **while** e **for** condividono il prezioso attributo di verificare la condizione di uscita all'inizio invece che alla fine del ciclo, come abbiamo visto nel Capitolo 1. Il terzo ciclo del C, **do-while**, la verifica alla fine dopo essere passato attraverso il corpo del ciclo; il corpo viene sempre eseguito almeno una volta. La sintassi è

do
    istruzione
while (*espressione*);

L'*istruzione* viene eseguita, poi viene valutata l'*espressione*. Se è vera, l'*istruzione* viene ancora eseguita e così via. Se l'espressione diventa falsa, il ciclo termina.
Come ci si potrebbe aspettare, **do-while** è molto meno usato di **while** e **for**; il suo uso contribuisce per il 5% tra quello di tutti i cicli. Ad ogni modo, è prezioso di tanto in tanto come nella seguente funzione **itoa**, che converte un numero in una stringa di caratteri (l'inverso di **atoi**). Il lavoro è leggermente più complicato di quanto si potrebbe pensare in un primo momento, poiché i metodi più semplici per generare le cifre le generano in un ordine sbagliato. Abbiamo scelto di generare la stringa al contrario e poi invertirla.

```
itoa(n, s)     /* converte n in caratteri all'interno di s */
char s[];
int n;
{
    int i, segno;
    if ((segno = n) < 0)     /* segno */
        n = -n;              /* rende n positivo */
    i = 0;
    do {         /* genera le cifre in ordine inverso */
        s[i++] = n % 10 + '0';     /* prende la cifra successiva */
    } while (( n /= 10) > 0);  /* la toglie */
    if (segno < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

Il **do-while** è necessario o perlomeno conveniente, poiché deve essere inserito almeno un carattere nell'array **s**, senza tener conto del valore di **n**. Inoltre abbiamo usato le graffe intorno all'unica istruzione che forma il corpo del **do-while**, anche se non sono necessarie, così il lettore frettoloso non scambia la parte **while** con l'*inizio* di un ciclo **while**.

Esercizio 3-3. In una rappresentazione numerica a complemento 2, la nostra versione di **itoa** non gestisce il più grosso numero negativo che è definito dal valore di **n** uguale a –(2^lunghezzaparola–1). Spiegare perché e modificarla in maniera tale che stampi il valore correttamente senza tener conto della macchina su cui gira.

Esercizio 3-4. Scrivere l'analoga funzione **itob (n, s)** che converte il valore assoluto intero **n** in una rappresentazione carattere-binaria in **s**. Scrivere **itoh**, che converte un intero in una rappresentazione esadecimale.

Esercizio 3-5. Scrivere una versione di **itoa** che accetti tre argomenti al posto di due. Il terzo argomento è la lunghezza minima del campo; se necesario, devono essere aggiunti dei blank a sinistra del numero convertito per ottenere l'ampiezza desiderata.

3.7 Break

Talvolta è comodo essere in grado di controllare l'uscita dal ciclo oltre che verificarla in testa o in coda. L'istruzione **break** fornisce una uscita prematura da **for, while** e **do**, come anche da uno **switch**. Un'istruzione **break** provoca l'uscita immediata dal ciclo più interno (o **switch**).
Il seguente programma rimuove i blank e tab finali da ogni linea dell'input, usando un **break** per uscire dal ciclo quando viene trovato il carattere più a destra diverso da blank e da tab.

```
#define     MAXLINEA        1000

main()     /* rimuove i blank ed i tab in coda */
{
     int n;
     char linea[MAXLINEA];

     while ((n = getline(linea, MAXLINEA)) > 0) {
          while (--n >= 0)
               if (linea[n] != ' ' && linea[n] != '\t'
                    && linea[n] != '\n')
                         break;
          linea[n+1] = '\0';
          printf("%s\n", linea);
     }
}
```

getline ritorna la lunghezza della linea. Il ciclo **while** più interno inizia all'ultimo carattere di **linea** (si tenga presente che **--n** decrementa **n** prima di usare il suo valore), e scandisce all'indietro alla ricerca del primo carattere che non è blank, tab o newline. Il ciclo viene spezzato quando se ne trova uno o quando n diviene negativo (per cui quando è stata scandita l'intera linea). Bisognerebbe verificare il suo corretto funzionamento anche quando la linea contiene solo spazi bianchi.
Un'alternativa a **break** è di mettere la verifica nel ciclo stesso:

```
while ((n = getline(linea, MAXLINEA)) > 0)  {
    while (--n >= 0
        && (linea[n]==' ' || linea[n]=='\t' || linea[n]=='\n'))
            ;

}
```

Questa versione è meno valida della precedente poiché il test è più difficile da capire. I test che richiedono una mescolanza di **&&, ||, !** o parentesi, generalmente non dovrebbero essere usati.

3.8 Continue

L'istruzione **continue** è dello stesso tipo di **break**, ma è usata meno frequentemente. Causa l'inizio della *successiva iterazione* del ciclo in cui è racchiusa **(for, while, do)**. In **while** e **do**, ciò significa che la parte di test viene immediatamente eseguita; nel **for** il controllo passa alla fase di reinizializzazione. (**continue** si applica solamente ai cicli, non a **switch**). Un **continue** all'interno di uno **switch** dentro un ciclo provoca la successiva iterazione del ciclo).
Come esempio, questo frammento opera solamente sugli elementi positivi dell'array **a**; i valori negativi vengono saltati.

```
for (i = 0; i < N; i++)  {
    if (a[i] < 0)     /* salta gli elementi negativi */
        continue;
    ...  /* elementi positivi */
}
```

L'istruzione **continue** viene spesso usata quando la parte del ciclo che segue è complicata, in maniera tale che invertendo un test e indentando a un altro livello, nidificherebbe il programma troppo profondamente.

Esercizio 3-6. Scrivere un programma che copia il proprio input sull'output, eccetto il fatto che stampa solamente una di ogni gruppo di linee adiacenti identiche. (Questa è una semplice versione dell'utility **UNIX** *uniq*).

3.9 Goto e Label

Il C fornisce un'istruzione **goto** di cui si può abusare, ed etichette a cui poter saltare. Formalmente il **goto** non è necessario, ed in pratica è quasi sempre facile scrivere un codice senza di esso. Non abbiamo usato **goto** in questo libro.
Ad ogni modo, suggeriremo alcune situazioni in cui si possono collocare i **goto**. L'uso più comune è quello di abbandonare le operazioni all'interno di qualche struttura profondamente nidificata, per esempio uscire da due cicli in una volta sola. L'istruzione **break** non può essere usata direttamente poiché fa uscire solamente dal ciclo più interno. Così:

```
        for ( ... )
            for ( ... ) {
                ...
                if (disastro)
                    goto errore;
            }
        ...

    errore:
     rimedia al pasticcio
```

Questa struttura è utile se la parte che gestisce gli errori è complessa e se gli errori possono avvenire in più parti.
Un'etichetta ha la stessa forma del nome di una variabile ed è seguita da un due punti. Può essere applicata ad ogni istruzione nella medesima funzione in cui si trova il **goto**. Come altro esempio, consideriamo il problema di trovare il primo elemento negativo in un array a due dimensioni (gli array a più dimensioni sono trattati nel Capitolo 5). Una possibilità è

```
        for (i = 0; i < N; i++)
            for (j = 0; j < M; j++)
                if (v[i][j] < 0)
                    goto trovato;
        /* non e' stato trovato */
        ...
    trovato:
        /* ne e' stato trovato uno alla posizione i, j */
        ...
```

Un codice che usi un **goto** può sempre essere scritto senza di esso, benché forse al prezzo di qualche ripetizione di test o di una variabile in più. Per esempio, la ricerca nell'array diventa

```
    trovato = 0;
    for (i = 0; i < N && !trovato; i++)
        for (j = 0; j < M && !trovato; j++)
            trovato = v[i][j] < 0;
    if (trovato)
        /* era alla posizione i-1, j-1 */
        ...
    else
        /* non trovato */
        ...
```

Benché non siamo dogmatici sull'argomento, a nostro parere l'istruzione **goto** dovrebbe essere usata con parsimonia o non usata del tutto.

CAPITOLO 4

FUNZIONI E STRUTTURA DEI PROGRAMMI

Le funzioni suddividono grossi impegni di calcolo in parti più semplici e permettono al programmatore di costruire su ciò che altri hanno fatto invece di iniziare da zero. Funzioni appropriate possono spesso celare i dettagli a quelle parti di programma a cui non è necessario conoscerli; ne deriva maggior chiarezza dell'insieme e una riduzione dello sforzo nell'apportare modifiche.
Il C è stato progettato per rendere le funzioni efficienti e facili da usare; i programmi in C consistono generalmente di funzioni piccole e numerose invece che di poche e grandi. Un programma può essere contenuto in uno o più files sorgenti nel modo più conveniente; i files sorgenti possono essere compilati separatamente e caricati insieme, ed anche con funzioni di librerie precedentemente compilate. Non ci inoltreremo ora in questo procedimento, dato che i dettagli variano a seconda del sistema. La maggior parte dei programmatori è abituata alle funzioni di libreria per l'input e l'output (**getchar, putchar**) ed alle operazioni numeriche (**sin, cos, sqrt**). In questo capitolo illustreremo più diffusamente ciò che riguarda la scrittura di nuove funzioni.

4.1 Fondamenti

Per iniziare, progettiamo e scriviamo un programma che stampi le righe di input che contengano un particolare "modello" (pattern) o stringa di caratteri. (Questo è un caso speciale del programma di utilità di UNIX *grep*). Ad esempio, ricercando il modello "the" nel gruppo di righe

```
When she gets there she knows
if the stores are all closed
with a word she can get
what she came for     .
```

produrrà l'output

```
When she gets there she knows
if the stores are all closed
```

La struttura base del lavoro si divide palesemente in tre parti:

while (*c'è un'altra linea*)
  if (*la linea contiene il modello cercato*)
    stampala

Anche se è certamente possibile codificare il programma in modo tale da fare tutto questo nella routine principale, un modo migliore è l'uso della struttura naturale così da progredire creando per ciascuna parte una funzione separata. Tre piccole parti sono più semplici da trattare di quanto lo sia una sola grossa, dato che i dettagli irrilevanti possono venire relegati nelle funzioni, riducendo la possibilità di interazioni non volute. Oltretutto le parti possono persino essere utili di per se stesse.
"Finché c'è un'altra linea" è **getline**, una funzione che scrivemmo nel Capitolo 1, e "stampala" è **printf**, che qualcuno ci ha già messo a disposizione. Ciò significa che a noi basta scrivere una routine che decida se la linea contenga una ricorrenza del modello. Possiamo risolvere il problema appropriandoci di una costruzione tratta da **PL/1**: la funzione **index(s, t)** ritorna la posizione o indice nella stringa **s** dove inizia la stringa **t**, o **−1** se **s** non contiene **t**. Usiamo 0 invece di 1 come posizione di partenza in **s** dato che gli array del C cominciano alla posizione 0. Quando più avanti avremo bisogno di una ricerca più sofisticata non faremo altro che sostituire **index**; il resto del codice può rimanere invariato.
Dato questo progetto di massima, il dettaglio del programma seguirà di conseguenza. Ecco di seguito tutto il programma, in modo da potervi rendere conto di come i vari pezzi vadano a posto fra di loro. Per ora il modello che deve essere cercato è una stringa letterale nell'argomento di **index**, che non è il meccanismo più generale possibile. Avremo modo tra breve di riconsiderare il problema di come inizializzare array di caratteri, e nel Capitolo 5 mostreremo come fare del modello un parametro dato al momento dell'esecuzione del programma. Questa è inoltre una nuova versione di **getline**; potrebbe essere istruttivo paragonarla a quella del Capitolo 1.

```
#define MAXLINEA  1000

main()   /* trova tutte le linee che contengono una stringa */
{
    char linea[MAXLINEA];

    while(getline(linea, MAXLINEA) > 0)
        if (index(linea, "the") >= 0)
            printf("%s", linea);
}
```

```
getline(s, lim)  /* legge la linea in s, ritorna la lunghezza */
char s[];
int lim;
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return(i);
}

index(s, t)   /* ritorna l'indice di t in s oppure -1 se t non c'e' */
char s[], t[];
{
    int i, j, k;

    for (i = 0; s[i] != '\0'; i++) {
        for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
            ;
        if (t[k] == '\0')
            return(i);
    }
    return(-1);
}
```

Ogni funzione ha la forma

nome (elenco di argomenti, se ce ne sono)
dichiarazione degli argomenti, se ce ne sono
{
dichiarazioni e istruzioni, se ce ne sono
}

Come suggerito, le varie parti possono essere assenti; una funzione minimale è

```
nulla()  { }
```

che non fa nulla. (Una funzione che non fa niente può essere qualche volta utile come "occupa-posizione" durante lo sviluppo del programma). Il nome della funzione può essere inoltre preceduto da un tipo se la funzione ritorna qualcosa di diverso da un valore intero; questo sarà l'argomento del prossimo paragrafo.
Un programma non è altro che un set di definizioni di funzioni individuali. La comunicazione fra le funzioni è (in questo caso) ottenuta tramite argomenti e valori ritornati dalle funzioni; ciò può anche verificarsi attraverso variabili esterne. Le funzioni possono apparire in qualsiasi ordine nel file sorgente, e il programma sorgente può essere diviso in più files, a condizione che nessuna funzione venga spezzata.

L'istruzione **return** è il meccanismo per restituire un valore dalla funzione che è stata chiamata alla funzione chiamante. Qualsiasi espressione può far seguito a **return**:

return (*espressione*)

La funzione che chiama è libera di ignorare il valore di ritorno se lo desidera. Inoltre non c'è bisogno che ci sia alcuna espressione dopo **return**; quando l'esecuzione trova la fine della funzione nel raggiungere la graffa chiusa, il controllo ritorna alla funzione chiamante senza alcun valore. Non è illecito, ma probabilmente è un'indicazione di pericolo, se una funzione ritorna un valore da una posizione e nessuno dall'altra. In ogni caso, il "valore" di una funzione che non ne ritorna alcuno è indefinito. Il verificatore del C *lint* serve per il controllo di tali errori.
I meccanismi di compilazione e caricamento di un programma in C che risiede su più files sorgenti variano da un sistema all'altro. Nel sistema **UNIX**, ad esempio, il comando *cc* citato nel Capitolo 1 provvede al compito. Si supponga che le tre funzioni siano su tre files chiamati *main.c, getline.c* e *index.c*. Quindi il comando

cc main.c getline.c index.c

compila i tre files, colloca il codice oggetto rilocabile nei files *main.o getline.o* e *index.o* e li carica tutti in un file eseguibile chiamato *a.out*.
Se c'è un errore, ad esempio in *main.c*, il file può essere ricompilato a parte e il risultato caricato con i precedenti file oggetto col comando

cc main.c getline.o index.o

Il comando *cc* usa il suffisso *".c"* in contrasto a *".o"* convenzionalmente per distinguere i file sorgenti dai file oggetto.

Esercizio 4-1. Scrivere la funzione **rindex(s, t)**, che restituisce la posizione della ricorrenza *più a destra* di **t** in **s**, o **−1** se non ce n'è alcuna.

4.2 Funzioni che non ritornano interi

Fino a questo momento, nessuno dei nostri programmi ha contenuto qualche dichiarazione del tipo di una funzione. Questo perché per default una funzione è implicitamente dichiarata dalla propria presenza in un'espressione o istruzione, come

```
while (getline(linea, MAXLINEA) > 0)
```

Se un nome che non è stato precedentemente dichiarato ricorre in un'espressione ed è seguito da una parentesi aperta, è dichiarato dal contesto come nome di funzione. Inoltre per default si presume che la funzione riporti un **int**. Dato che **char** si eleva ad **int** nelle espressioni, non c'è bisogno di dichiarare le funzioni che restituiscono **char**.

Questi assunti si riferiscono alla maggior parte dei casi, includendo tutti gli esempi finora riportati.
Ma che succede se una funzione deve restituire qualche altro tipo? Molte funzioni numeriche come **sqrt, sin,** e **cos** ritornano un valore di tipo **double**; altre funzioni specializzate ritornano altri tipi. Ad illustrare come ci si comporta in questi casi, scriviamo e usiamo la funzione **atof(s)**, che converte la stringa s nel proprio equivalente floating point in doppia precisione. **atof** è un'estensione di **atoi**, di cui scrivemmo alcune versioni nei Capitoli 2 e 3; essa tratta anche un segno opzionale ed un punto decimale, e la presenza o assenza della parte intera o decimale. (*Non* si tratta di una routine di conversione dell'input di alta qualità; in quel caso si richiederebbe più spazio di quel che ci preme usare.)
Per prima cosa, **atof** stessa deve dichiarare il tipo di valore ritornato, poiché non è **int**. Dato che nelle espressioni **float** è convertito in **double**, non c'è ragione di affermare che **atof** restituisca **float**; potremmo anche far uso della maggiore precisione e perciò dichiariamo che ritorna double. Il nome del tipo precede il nome della funzione in questo modo:

```
double atof(s)  /* converte la stringa s in double */
char s[];
{
    double val, potenza;
    int i, segno;

    for (i=0; s[i]==' ' || s[i]=='\n' || s[i]=='\t'; i++)
        ;   /* salta gli spazi bianchi */
    segno = 1;
    if (s[i] == '+' || s[i] == '-')   /* segno */
        segno =(s[i++]=='+') ? 1 : -1;
    for (val = 0; s[i] >= '0' && s[i] <='9'; i++)
        val = 10 * val + s[i] - '0';
    if (s[i] == '.')
        i++;
    for (potenza = 1; s[i] >= '0' && s[i] <= '9'; i++) {
        val = 10 * val + s[i] - '0';
        potenza *= 10;
    }
    return(segno * val / potenza);
}
```

Secondariamente, ma ugualmente importante, la routine *chiamante* deve dichiarare che **atof** restituisce un valore non **int**. La dichiarazione è mostrata nel seguente primitivo calcolatore da tavolo (a stento adeguato per un bilancio di prima nota), che legge un numero per linea preceduto opzionalmente da un segno, e ne fa il totale stampando la somma dopo ogni input.

```
#define  MAXLINEA  100

main()  /* rudimentale calcolatore da tavolo */
{
    double somma, atof();
    char linea[MAXLINEA];

    somma = 0;
    while (getline(linea, MAXLINEA) > 0
        printf("\t%.2f\n", somma += atof(linea));
}
```

La dichiarazione

```
double somma, atof();
```

dice che **somma** è una variabile **double** e che **atof** è una funzione che restituisce un valore **double**. Come mnemonico, suggerisce che **somma** e **atof (...)** sono entrambi valori floating point in doppia precisione.
A meno che **atof** sia esplicitamente dichiarata in entrambi i posti, il C presume che venga restituito un numero intero, e si otterranno quindi risposte senza senso. Se alla stessa **atof** ed alla relativa chiamata in **main** vengono attribuiti tipi incompatibili nello stesso file sorgente, ciò sarà riscontrato dal compilatore. Ma se (come più probabile) **atof** fosse compilata separatamente, il conflitto non verrebbe rilevato, **atof** restituirebbe un **double** che **main** tratterebbe come un **int**, e ne risulterebbero risposte senza significato. (*lint* rileva errori di questo tipo).
Data **atof**, potremmo in linea di principio scrivere **atoi** (convertire una stringa in **int**) nei suoi termini:

```
atoi(s)  /* converte la stringa s in intero */
char s[];
{
    double atof();

    return(atof(s));
}
```

Notare la struttura delle dichiarazioni e l'istruzione **return**. Il valore dell'espressione in

return (*espressione*)

è sempre convertito nel tipo della funzione prima che avvenga il ritorno. Perciò,il valore di **atof**, un **double**, è convertito automaticamente in un **int** quando appare in un **return**, dato che la funzione **atoi** restituisce un **int**. (La conversione di un valore floating point in **int** tronca qualsiasi parte decimale, così come discusso nel Capitolo 2).

Esercizio 4-2. Estendere **atof** così che tratti notazioni scientifiche della forma

```
123.45e-6
```

in cui un numero floating point possa essere seguito da **e** oppure **E** e da un'opzionale esponente con segno.

4.3 Ulteriori precisazioni sugli argomenti di funzione

Nel Capitolo 1 abbiamo discusso il fatto che gli argomenti delle funzioni vengono passati per valore, cioè la funzione chiamata riceve una copia privata e temporanea di ogni

argomento, non il suo indirizzo. Questo significa che la funzione non può modificare l'argomento originale nella funzione che chiama. Nell'ambito di una funzione, ogni argomento è una variabile locale inizializzata col valore con il quale la funzione è stata chiamata.
Quando un nome di array appare come argomento di una funzione, viene passata la locazione di inizio dell'array; gli elementi non vengono copiati. La funzione può alterare elementi dell'array indicizzando da questa posizione. L'effetto è che gli array vengono passati per indirizzo. Nel Capitolo 5 discuteremo l'uso dei puntatori per permettere alle funzioni di modificare le variabili non vettoriali delle funzioni chiamanti.
Fra le altre cose, non esiste un modo completamente soddisfacente per scrivere una funzione portabile che accetti un numero variabile di argomenti, poiché per la funzione chiamata non esiste un meccanismo portabile per determinare quanti argomenti fossero di fatto passati alla stessa in una data chiamata. Perciò, non si può scrivere una funzione veramente portabile che calcoli il massimo di un numero arbitrario di argomenti così come fanno le funzioni **MAX** incorporate nel Fortran e nel **PL/1**.
In generale è sicuro trattare un numero variabile di argomenti se la funzione richiamata non usa gli argomenti che non siano stati effettivamente procurati, e se i tipi sono compatibili. **printf**, la più comune funzione del C con un numero variabile di argomenti, usa le informazioni del primo argomento per determinare quanti altri argomenti sono presenti e quali sono i loro tipi. Non ha alcun risultato se il chiamante non fornisce argomenti sufficienti o se i tipi non corrispondono a quello che è indicato dal primo argomento. Non è inoltre portabile e dev'essere modificata per differenti ambienti.
Alternativamente, se gli argomenti sono di tipo conosciuto è possibile contrassegnare la fine della lista di argomenti in qualche maniera predeterminata, quale uno speciale valore dell'argomento (spesso 0) che rappresenta la fine degli argomenti.

4.4 Variabili Esterne

Un programma C consiste di un insieme di oggetti esterni rappresentati sia dalle variabili che dalle funzioni. L'aggettivo "esterno" viene soprattutto usato in contrasto a "interno", che descrive gli argomenti e le variabili automatiche definite all'interno delle funzioni. Le variabili esterne vengono definite al di fuori di ogni funzione e perciò sono potenzialmente disponibili a molte funzioni. Le funzioni stesse sono sempre esterne per il fatto che il C non permette che le funzioni siano definite all'interno di altre funzioni. Pe default, le variabili esterne sono anche "globali" in modo che tutti i riferimenti ad una certa variabile per mezzo dello stesso nome (anche per le funzioni compilate separatamente) sono riferimenti allo stesso oggetto. In questo senso, le variabili esterne sono analoghe al **COMMON** del Fortram o all'**EXTERNAL** del **PL/1**. Vedremo più avanti come definire variabili esterne e funzioni che non sono globalmente disponibili ma che sono invece visibili solo all'interno di un singolo file sorgente.
Per il fatto che le variabili esterne sono globalmente accessibili, sono un'alternativa agli argomenti di funzione ed ai valori di ritorno per la comunicazione dei dati tra le funzioni. Ogni funzione può accedere ad una variabile esterna riferendosi ad essa attraverso il nome, se il nome è già stato da qualche parte dichiarato.
Se le funzioni devono condividere un grande numero di variabili, le variabili esterne sono più convenienti ed efficienti di un lungo elenco di argomenti. Comunque, come sottolineato nel Capitolo 1, questo metodo dev'essere applicato con alcune precauzioni

poiché potrebbe portare ad un programma mal strutturato e con troppe connessioni di dati tra le funzioni.
Una seconda ragione per usare le variabili esterne riguarda l'inizializzazione. In particolare, a differenza degli array automatici, gli array esterni possono essere inizializzati. Tratteremo l'inizializzazione verso la fine di questo capitolo.
La seconda ragione per usare le variabili esterne riguarda la loro visibilità e ciclo di vita. Le variabili automatiche sono interne ad una funzione; esistono solamente nel periodo di tempo tra l'entrata e l'uscita da una funzione. Invece le variabili esterne sono permanenti. Esse non vanno e vengono per cui mantengono i valori tra una chiamata di funzione e l'altra. Quindi se due funzioni hanno bisogno di condividere alcuni dati, e nessuna delle due chiama l'altra, è generalmente conveniente che i dati condivisi si trovino in variabili esterne invece di essere passati dentro e fuori attraverso gli argomenti.
Vediamo meglio questa problematica con un esempio più significativo. Il problema è quello di scrivere un altro programma calcolatore, migliore del precedente. Questo permette le operazioni +, −, *, / e = (per stampare la riposta). Poiché è per alcuni versi più semplice da implementare, il calcolatore userà la notazione Polacca inversa al posto della infissa. (Questo schema viene usato per esempio dai calcolatori tascabili Hewlett-Packard). Nella notazione Polacca Inversa, ogni operatore segue i propri operandi; una notazione infissa del tipo

```
(1 - 2) * (4 + 5) =
```

viene inserita come

```
1 2 - 4 5 + * =
```

Le parentesi non sono necessarie.
L'implementazione è piuttosto semplice. Ogni operando viene inserito in uno stack (operazione di push); quando arriva un operatore, vengono sganciati (operazioni di pop) un numero adeguato di operandi (due per gli operatori binari), si applica ad essi l'operatore ed il risultato reinserito nello stack. Nell'esempio precedente, per esempio, vengono inseriti 1 e 2, rimpiazzati poi dalla loro differenza, −1. Successivamente vengono inseriti 4 e 5, rimpiazzati dalla loro somma, 9. Il prodotto di −1 e 9, che è −9, li rimpiazza nello stack. L'operatore = stampa l'elemento in testa senza rimuoverlo (così in un calcolo si possono verificare passi intermedi).
Le operazioni di inserire ed estrarre uno stack sono banali, ma nel momento in cui sono comprese le segnalazioni ed il ripristino da errore, sono lunghe abbastanza per preferire di collocarle in funzioni separate invece di ripeterne il codice in tutto il programma. Dovrebbe anche esserci una funzione separata per accettare il successivo operatore o operando in input. Quindi la struttura del programma è

while (*il prossimo operatore o operando non è la fine del file*)
  if (*è un numero*)
    inseriscilo
  else if (*è un operatore*)
    estrai gli operandi
    esegui le operazioni
    inserisci il risultato
  else
    errore

La principale decisione della struttura che non è ancora stata trattata è su dove si trovi lo stack, cioè quali routine accedono ad esso direttamente. Una possibilità è di mantenerlo in **main** e di passare lo stack e la posizione corrente all'interno di esso alle routine che lo inseriscono e lo estraggono. Però **main** non ha bisogno di conoscere le variabili che controllano lo stack; dovrebbe occuparsi solo di inserimento ed estrazione. Quindi abbiamo deciso di mantenere lo stack e le informazioni ad esso associate nelle funzioni **push** e **pop** e non all'interno di **main**.
Tradurre questo schema è piuttosto semplice. Fondamentalmente il programma principale è un grande **switch** sul tipo degli operatori od operandi; forse questo è un uso più tipico di **switch** di quello visto nel Capitolo 3.

```
#define MAXOP  20   /* massima ampiezza di operandi e operatori */
#define NUMERO '0'  /* segnala che e' stato trovato un numero */
#define TROPPO '9'  /* segnala che la stringa e' troppo grande */

main()   /* calcolatore da tavolo in notazione Polacca Inversa */
{
    int tipo;
    char s[MAXOP];
    double op2, atof(), pop(), push();

    while((tipo = getop(s, MAXOP)) != EOF)
        switch (tipo)  {

        case NUMERO:
            push(atof(s));
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - pop());
            break;
        case '/':
            op2 = pop();
            if (op2 != 0.0)
                push(pop() / op2);
            else
                printf("estratto un divisore uguale a zero\n");
            break;
        case '=':
            printf("\t%f\n",push(pop()));
            break;
        case 'c':
            clear();
            break;
        case TROPPO:
            printf("%.20s ... e' troppo lunga\n", s);
            break;
        default:
            printf("comando sconosciuto %c\n", tipo);
            break;
        }
}
```

```
#define  MAXVAL  100  /* profondita' massima dello stack val */

int sp = 0;          /* puntatore allo stack */
double val[MAXVAL];  /* stack dei valori */

double push(f)  /* inserisce f nello stack di valori */
double f;
{
    if (sp < MAXVAL)
        return(val[sp++] = f);
    else {
        printf("errore: stack pieno\n");
        clear();
        return(0);
    }
}

double pop()  /* estrae il valore dall'alto dello stack */
{
    if (sp > 0)
        return(val[--sp]);
    else {
        printf("errore: stack vuoto\n");
        clear();
        return(0);
    }
}

clear()   /* ripulisce lo stack */
{
    sp = 0;
}
```

Il comando **c** ripulisce lo stack attraverso la funzione **clear** che viene anche usata da **push** e **pop** in caso di errore. Ritorneremo a **getop** fra un momento.
Come visto nel Capitolo 1, una variabile è esterna se viene definita al di fuori del corpo di ogni funzione. Per questo motivo lo stack ed il puntatore allo stack che devono essere condivisi da **push, pop** e **clear** sono definiti fuori da queste tre funzioni. Ma **main** stesso *non* si riferisce né allo stack né al puntatore allo stack — la rappresentazione è cautamente celata. Quindi il codice per l'operatore = deve usare

```
push(pop());
```

per esaminare la cima dello stack senza modificarlo.
Da notare inoltre che siccome + e * sono operatori commutativi, è irrilevante l'ordine con cui vengono combinati gli operandi estratti, ma per gli operatori − e / gli operandi di sinistra e di destra devono essere distinti.

Esercizio 4-3. Data l'architettura basilare, l'estensione del calcolatore viene di conseguenza. Aggiungere gli operatori di modulo **(%)** ed il segno meno unario. Aggiungere un comando di "cancellazione" che cancelli l'entrata sulla cima dello stack. Aggiungere comandi per la gestione delle variabili. (È facile con nomi di variabili di una sola lettera).

4.5 Regole di Visibilità

Le funzioni e le variabili esterne che compongono un programma C non necessitano di essere compilate contemporaneamente; si può mantenere il testo del programma in parecchi files e caricare ruotine già compilate da librerie. Due interessanti domande sono:

Come vengono scritte le dichiarazioni in maniera che le variabili siano correttamente dichiarate durante la compilazione?
Come vengono collocate le dichiarazioni in maniera che tutti i pezzi siano correttamente collegati quando il programma viene caricato?

La *visibilità* di un nome è la parte del programma in cui è definito quel nome. Per una variabile automatica dichiarata all'inizio di una funzione, la visibilità è la funzione in cui ne è dichiarato il nome, e non ha alcuna relazione con variabili dello stesso nome di altre funzioni. Ciò è vero anche per gli argomenti di funzione.
La visibilità di una variabile esterna va dal punto in cui è dichiarata in un file sorgente alla fine dello stesso. Per esempio, se **val, sp, push, pop** e **clear** sono definite in un file nell'ordine che vediamo qui sotto, cioè

```
int sp = 0;
double val[MAXVAL];

double push(f)  { ... }

double pop()  { ... }

clear()  { ... }
```

allora le variabili **val** e **sp** possono essere usate all'interno di **push, pop** e **clear** richiamandolo semplicemente; non sono necessarie ulteriori dichiarazioni.
Dall'altra parte, se bisogna riferirsi ad una variabile esterna prima che sia definita o se è definita in un file sorgente *differente* da quello in cui sta per essere usata, è obbligatoria una dichiarazione **extern**.
È importante la distinzione tra la *dichiarazione* di una variabile esterna e la sua *definizione*. Una dichiarazione annuncia le caratteristiche di una variabile (il suo tipo, ampiezza, ecc.); una definizione fa anche in modo che venga allocata memoria. Se le linee

```
int sp;
double val[MAXVAL];
```

appaiono al di fuori di ogni funzione, *definiscono* le variabili esterne **sp** e **val**, allocano la memoria, e servono inoltre come dichiarazione per il resto del file sorgente. D'altro lato, le linee

```
extern int sp;
extern double val[];
```

dichiarano che **sp** è un **int** e che **val** è un array **double** (la cui ampiezza viene determinata da un'altra parte) per il resto del file sorgente, ma esse non creano le variabili e non allocano memoria per esse.
Ci può essere solamente una *definizione* di una variabile esterna tra tutti i files che costituiscono il programma sorgente; altri files possono contenere dichiarazioni **extern** per poterla accedere. (Può anche esserci una dichiarazione **extern** nel file che contiene la definizione). L'inizializzazione di una variabile esterna si può solamente ottenere insieme alla definizione. Le ampiezze degli array devono essere specificate con la definizione, ma sono opzionali con una dichiarazione **extern**.
Sebbene non è una utile organizzazione per questo programma, **val** e **sp** potrebbero essere definite e inizializzate in un file, e le funzioni **push, pop** e **clear** definite in un altro. Quindi è necessario che queste definizioni e dichiarazioni siano legate tra di loro.

```
Nel file 1:

    int sp = 0;            /* puntatore allo stack */
    double val[MAXVAL];    /* stack di valori */

Nel file 2:

    extern int sp;
    extern double val[];

    double push(f)  { ... }

    double pop()  { ... }

    clear()  { ... }
```

Poiché le dichiarazioni **extern** all'interno di *file 2* si collocano prima e fuori dalle tre funzioni, si applicano a tutte; per tutto *file 2* è sufficiente un'unica parte di dichiarazioni.
Per i programmi più estesi, la facility di inclusione di file **#include** (verrà trattata in questo capitolo) permette di mantenere un'unica copia delle dichiarazioni **extern** per il programma e fare sì che esse siano inserite in ciascun file sorgente al momento della compilazione.
Torniamo ora all'implementazione di **getop**, la funzione che accetta il successivo operatore o operando. L'idea basilare è semplice: salta i blank, tab e newline.
Se il successivo carattere non è una cifra o un punto decimale, viene ritornato. Altrimenti viene formata una stringa di cifre (che potrebbe contenere un punto decimale) e si ritorna **NUMERO**, il segnale che è stato trovato un numero.
La routine risulta piuttosto complicata perché gestisce correttamente la situazione che si verifica quando un numero di input è troppo grande. **getop** legge le cifre (magari con un punto decimale) finché le trova, ma immagazzina solo quelle che sono idonee. Se non si è verificato l'overflow, ritorna **NUMERO** e la stringa di cifre. Se il numero era troppo grande, comunque, getop ignora il resto della linea di input cosicché l'utente può semplicemente ribattere la linea dal punto di errore; come segnale di overflow ritorna **TROPPO**.

```
getop(s, lim)  /* prende il prossimo operatore od operando */
char s[];
int lim;
{
    int i, c;

    while ((c = getch()) == ' ' || c == '\t' || c == '\n')
        ;
    if (c != '.' && (c < '0' || c > '9'))
        return(c);
    s[0] = c;
    for (i = 1; (c = getchar()) >= '0' && c <= '9'; i++)
        if (i < lim)
            s[i] = c;
    if (c == '.')  {  /* parte decimale */
        if (i < lim)
            s[i] = c;
        for (i++; (c=getchar()) >= '0' && c = '9'; i++)
            if (i < lim)
                s[i] = c;
    }
    if (i < lim)  {  /* il numero e' corretto */
        ungetch(c);
        s[i] = '\0';
        return(NUMERO);
    } else {  /* e' troppo grande; salta il resto della linea */
        while (c != '\n' && c != EOF)
            c = getchar();
        s[lim-1] = '\0';
        return(TROPPO);
    }
}
```

Cosa sono **getch** e **ungetch**? Si verifica spesso che un programma che sta leggendo l'input non riesca a determinare che ha letto a sufficienza fino a che non ha letto troppo. Un esempio di ciò è raggruppare i caratteri che costituiscono un numero: fino a che non si trova il primo carattere che non sia una cifra, il numero non è completo.
Ma a questo punto il programma ha letto un carattere in più, un carattere per cui non è preparato.
Il problema sarebbe risolto se fosse possibile annullare la lettura del carattere non voluto. Così ogni volta che il programma legge un carattere di troppo può rimandarlo indietro nell'input così il resto del programma si può comportare come se non fosse mai stato letto. Fortunatamente è facile simulare l'annullamento di un carattere scrivendo una coppia di funzioni tra loro collegate. **getch** riporta il prossimo carattere di input che dev'essere considerato; **ungętch** rimanda un carattere indietro nell'input, in maniera che la prossima chiamata a **getch** lo ritornerà ancora.
Il modo in cui interagiscono è semplice. **ungetch** inserisce i caratteri rimandati indietro in un buffer condiviso, un array di caratteri. **getch** legge dal buffer se c'è dentro qualcosa; se il buffer è vuoto chiama **getchar**. Deve anche esserci una variabile che tenga l'indice della posizione del carattere corrente all'interno del buffer.
Poiché sia il buffer che l'indice sono condivisi da **getch**e **ungetch** e devono mantenere i loro valori tra le chiamate, devono essere esterni ad entrambe le routine. Possiamo quindi scrivere **getch** e **ungetch** e le loro variabili comuni:

```
#define  BUFAMPIO  100

char buf[BUFAMPIO];   /* buffer per ungetch */
int bufp = 0;  /* prossima posizione libera in buf */

getch()   /* prende un (magari rimandato indietro) carattere */
{
    return((bufp > 0) ? buf[--bufp] : getchar());
}

ungetch(c)  /* rimanda indietro un carattere nell'input */
int c;
{
    if (bufp > BUFAMPIO)
        printf("ungetch: troppi caratteri\n");
    else
        buf[bufp++] = c;
}
```

Per rimandare indietro abbiamo usato un array invece che un singolo carattere, poiché la generalità ci potrà venire utile più avanti.

Esercizio 4-4. Scrivere una routine **ungets(s)** che rimanderà indietro nell'input un'intera stringa. **ungets** dovrebbe conoscere **buf** e **bufp** o potrebbe solamente usare **ungetch**?

Esercizio 4-5. Supponiamo che non ci sarà mai più di un carattere da rimandare indietro. Modificare **getch** e **ungetch** di conseguenza.

Esercizio 4-6. Le nostre **getch** e **ungetch** non prendono in considerazione che **EOF** sia rimandato indietro in maniera portabile. Decidere quali proprietà ci dovrebbero essere se viene rimandato indietro **EOF**, implementando poi il proprio progetto.

4.6 Variabili static

Le variabili static sono la terza classe di memorizzazione, oltre alle **extern** e automatiche che abbiamo già incontrato.
Le variabili **static** possono essere sia interne che esterne. Le variabili **static** interne sono locali ad una particolare funzione come lo sono le variabili automatiche ma, a differenza di queste ultime, rimangono in esistenza invece che andare e venire ogni volta che viene attivata la funzione. Questo significa che le variabili **static** interne forniscono una memoria privata e permanente in una funzione. Le stringhe di caratteri che appaiono all'interno di una funzione, ad esempio come argomenti a **printf**, sono statiche interne.
Una variabile **static** esterna è nota all'interno della parte rimanente del *file sorgente* in cui è dichiarata e in nessun altro file. Le **static** esterne rappresentano un modo per nascondere nomi quali **buf** e **bufp** nelle combinazioni **getch-ungetch**, che devono essere esterni per poter essere condivisi, ma che non dovrebbero essere visibili agli

utenti di **getch** e **ungetch** in modo da non provocare alcun conflitto. Se le due routine e le due variabili vengono compilate in un file, come in

```
static char   buf[BUFAMPIO];  /* buffer per ungetch */
static int    bufp = 0;   /* successiva posizione libera in buf */

getch() { ... }

ungetch(c) { ... }
```

allora nessun'altra routine sarà in grado di accedere a **buf** e **bufp**; di fatto non creeranno conflitti con gli stessi nomi in altri files dello stesso programma.
La memoria statica, sia interna che esterna, viene specificata precedendo la dichiarazione normale con la parola **static**. La variabile è esterna se è definita al di fuori di ogni funzione ed interna se definita all'interno.
Generalmente le funzioni sono oggetti esterni; i loro nomi sono noti globalmente. È comunque possibile dichiarare una funzione **static**; il suo nome sarà così sconosciuto al di fuori del file in cui è stata dichiarata.
Nel C, **"static"** non connota solamente la permanenza ma anche un aspetto di ciò che può essere chiamato "privato". Gli oggetti **static** interni sono conosciuti solamente all'interno di una funzione. Gli oggetti (variabili o funzioni) **static** esterni sono conosciuti solamente all'interno del file sorgente in cui appaiono ed i loro nomi non interferiscono con variabili o funzioni dello stesso nome in altri files.
Le variabili e le funzioni esterne **static** forniscono un modo per celare dati ed ogni routine interna che li manipola in modo che altre routine e dati non possono creare conflitti neanche inavvertitamente.
Per esempio, **getch** e **ungetch** formano un "modulo" per l'input e la respinta dei caratteri; **buf** e **bufp** per poter essere inaccessibili dall'esterno devono essere **static**. Nello stesso modo, **push pop** e **clear** formano un modulo per la manipolazione dello stack; anche **val** e **sp** devono essere **static** esterne.

4.7 Variabili Register

La quarta e ultima classe di memorizzazione è chiamata **register**. Una dichiarazione **register** avvisa il compilatore che la variabile in questione sarà usata in maniera molto intensa. Quando possibile, le variabili **register** sono collocate nei registri della macchina, per cui risultano programmi più sintetici e veloci.
La dichiarazione **register** appare come

```
register int  x;
register char c;
```

e così via; la parte **int** può essere omessa. **register** può essere applicato solamente

alle variabili automatiche ed ai parametri formali di una funzione. In quest'ultimo caso, la dichiarazione appare così:

```
f(c, n)
register int c, n;
{
    register int i;
    ...
}
```

Nella pratica, ci sono alcune restrizioni sulle variabili register che riflettono la realtà dell'hardware. Nei registri si può mantenere solo un piccolo numero di variabili per ogni funzione e sono permessi solo alcuni tipi. La parola **register** viene ignorata nel caso di dichiarazioni eccedenti o non permesse. Inoltre non è possibile conoscere l'indirizzo di una variabile register (argomento che sarà trattato nel Capitolo 5). Le specifiche restrizioni variano da una macchina all'altra; per esempio, sul **PDP-11** in una funzione sono effettive solo le prime tre dichiarazioni register ed i tipi devono essere **int, char** o puntatore.

4.8 Struttura a Blocchi

Il C non è un linguaggio strutturato a blocchi nel senso del **PL/1** o Algol, cioè le funzioni non possono essere definite all'interno di altre funzioni.
D'altra parte, le variabili possono essere definite in stile strutturato a blocchi. Le dichiarazioni di variabili (comprese le inizializzazioni) possono seguire la graffa aperta che introduce *qualsiasi* istruzione composta, non solamente quella che inizia una funzione. Le variabili dichiarate in questo modo si sovrappongono ad ogni identico nome di variabile nei blocchi più esterni e rimangono in esistenza fino alla graffa chiusa corrispondente. Per esempio, in

```
if (n > 0) {
    int i;     /* dichiara una nuova i */
    for (i = 0; i < n; i++)
        ...
}
```

la visibilità della variabile **i** è la parte "vera" dell'**if**; questa i non è collegata ad alcuna altra **i** nel programma.
La struttura a blocchi si applica anche alle variabili esterne. Con la dichiarazione

```
int x;

f()
{
    double x;
    ...
}
```

le occorrenze di **x** all'interno della funzione **f** si riferiscono alla variabile interna **double**;

all'esterno di **f** si riferiscono all'intero esterno. La stessa cosa è valida per i nomi dei parametri formali:

```
int z;

f(z)
double z;
{

}
```

All'interno della funzione **f, z** si riferisce al parametro formale, non alla variabile esterna.

4.9 Inizializzazione

L'inizializzazione è stata già citata parecchie volte, ma sempre di passaggio nella trattazione di altri argomenti. Ora che abbiamo già affrontato le varie classi di memorizzazione, questo paragrafo riassume alcune regole.
Nell'assenza di inizializzazioni esplicite, è garantito che le variabili esterne e statiche sono a zero; le variabili automatiche e register hanno valori indefiniti (quello che capita).
Le variabili semplici (non array o strutture) possono essere inizializzate quando vengono dichiarate facendone seguire il nome da un segno di uguale ed un'espressione costante:

```
int x = 1;
char apice = '\'';
long giorno = 60 * 24;  /* minuti in un giorno */
```

Per le variabili esterne e statiche l'inizializzazione viene effettuata una sola volta, concettualmente in fase di compilazione. Per le variabili automatiche e register viene effettuata ogni volta che si entra nella funzione o blocco.
Sempre per queste ultime, l'inizializzatore non è limitato ad una costante: di fatto può essere qualsiasi valida espressione che richiama valori precedentemente definiti, comprese le chiamate di funzione. Per esempio, le inizializzazioni del programma di ricerca binaria del Capitolo 3 potrebbero essere scritte come

```
binary(x, v, n)
int x, v[], n;
{
    int basso = 0;
    int alto = n - 1;
    int mezzo;
    ...
}
```

invece di

```
binary(x, v, n)
int x, v[], n;
{
    int basso, alto, mezzo;

    basso = 0;
    alto = n - 1;
    ...
}
```

Di fatto, le inizializzazioni delle variabili automatiche sono solamente abbreviazioni per istruzioni di assegnamento. Quale forma scegliere dipende solo dalle preferenze individuali. Generalmente noi abbiamo usato assegnamenti espliciti, poiché inizializzare nelle dichiarazioni è meno visibile.
Gli array automatici non possono essere inizializzati. Gli array esterni e statici possono essere inizializzati facendo seguire la dichiarazione da un elenco di inizializzatori racchiusi tra graffe e separati da virgole. Per esempio, il programma di conteggio caratteri del Capitolo 1, che inizia con

```
main()   /* conta cifre, spazi bianchi, altri */
{
    int c, i, nbianchi, naltri;
    int ncifre[10];

    nbianchi = naltri = 0;
    for (i = 0; i < 10; i++)
        ncifre[i] = 0;
    ...
}
```

può anche essere scritto come

```
int nbianchi = 0;
int naltri = 0;
int ncifre[10] ={ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

main()   /* conta cifre, spazi bianchi, altri */
{
    int c, i;
    ...
}
```

In realtà queste inizializzazioni non sono necessarie poiché le variabili sono già tutte a zero, ma è comunque un buon metodo renderle esplicite. Se ci sono meno valori iniziali dell'ampiezza specificata, i rimanenti saranno zero. È un errore avere troppi inizializzatori. Purtroppo, non c'è modo di specificare ripetizioni di un inizializzatore e neppure di inizializzare un elemento nel mezzo di un array senza dare tutti i valori precedenti.
Gli array di caratteri rappresentano un caso speciale delle inizializzazioni; si può usare una stringa al posto della notazione con le graffe e le virgole:

```
char modello[] = "the";
```

Questa è un'abbreviazione per la forma più lunga ma equivalente

```
char modello[] = { 't', 'h', 'e', '\0' };
```

Quando viene omessa l'ampiezza di un array di qualsiasi tipo, il compilatore ne calcolerà la lunghezza contando i valori iniziali. In questo caso particolare, l'ampiezza è 4 (tre caratteri più il terminatore \0).

4.10 Ricorsione

Le funzioni C possono essere usate ricorsivamente; cioè, una funzione può chiamare *se stessa* sia direttamente che indirettamente. Un tipico esempio riguarda la stampa di un numero come stringa di caratteri. Come già menzionato, le cifre vengono generate nell'ordine errato: le cifre con valore minore sono disponibili prima delle cifre con valore più alto, ma esse devono essere stampate nell'ordine inverso.
Ci sono due soluzioni a questo problema. Una è collocare le cifre in un array nello stesso modo in cui vengono generate e stamparle poi in ordine inverso, come facemmo nel Capitolo 3 con **itoa**. La prima versione di **printd** segue questo modello.

```
printd(n)    /* stampa n in decimale */
int n;
{
     char s[10];
     int i;

     if (n < 0) {
          putchar('-');
          n = -n;
     }
     i = 0;
     do {
          s[i++] = n % 10 + '0';  /* prende il prossimo carattere */
     } while ((n /= 10) > 0); /* lo abbandona */
     while (--i >= 0)
          putchar(s[i]);
}
```

L'alternativa è una soluzione ricorsiva in cui ogni chiamata a **printd** chiama prima se stessa per considerare le cifre in testa, poi stampa le cifre in coda.

```
printd(n)    /* stampa n in decimale (ricorsivo) */
int n;
{
     int i;

     if (n < 0) {
          putchar('-');
          n = -n;
     }
     if ((i = n/10) != 0)
          printd(i);
     putchar(n % 10 + '0');
}
```

Quando una funzione chiama se stessa ricorsivamente, ad ogni invocazione si ha un nuovo insieme di tutte le variabili automatiche, indipendente dal precedente. Così in **printd(123)** la prima **printd** ha **n** = **123**. Passa 12 ad una seconda **printd**, stampando poi 3 quando quella ritorna. Allo stesso modo, la seconda **printd** passa 1 ad una terza (che lo stampa), poi stampa 2.
Generalmente la ricorsione non provvede a risparmiare memoria per il fatto che

dev'essere mantenuto da qualche parte uno stack dei valori attivi. Non sarà neanche più veloce. Ma un codice ricorsivo è più compatto e spesso più semplice da scrivere e da capire. La ricorsione è particolarmente conveniente per strutture di dati definite ricorsivamente come gli alberi; ne vedremo un buon esempio nel Capitolo 6.

Esercizio 4-7. Adattare l'idea di **printd** per scrivere una versione ricorsiva di **itoa**; cioè, convertire un intero in una stringa con una routine ricorsiva.

Esercizio 4-8. Scrivere una versione ricorsiva della funzione **reverse(s)**, che inverte la stringa **s**.

4.11 Il Preprocessore C

Il C permette alcune estensioni del linguaggio attraverso un semplice preprocessore di macro. L'estensione più comune è data da #**define**, che abbiamo già usato; un'altra è rappresentata dalla capacità di includere i contenuti di altri files in fase di compilazione.

Inclusione di file

Per facilitare la disponibilità di un insieme di #**define** e dichiarazioni (fra le altre cose), il C fornisce una caratteristica di inclusione di file. Ogni linea della forma

#**include** *"nome-file"*

viene rimpiazzata dai contenuti del file *nome-file*. (I doppi apici sono obbligatori). Una o due linee di questa forma compaiono spesso all'inizio di ogni file sorgente, per includere istruzioni #**define** comuni e dichiarazioni **extern** per le variabili globali. I #**include** possono essere nidificati.
#**include** è il modo migliore per legare insieme le dichiarazioni in un programma esteso. Garantisce che tutti i file sorgenti avranno a disposizione le stesse definizioni e dichiarazioni di variabili, eliminando così un tipo di errore particolarmente nascosto. Naturalmente quando viene modificato un file da includere, andranno ricompilati tutti i files che dipendono da esso.

Sostituzione di macro

Una definizione della forma

```
#define   YES   1
```

richiede una sostituzione di macro del tipo più semplice:
sostituisce un nome con una stringa di caratteri. I nomi in #**define** si presentano con la stessa forma degli identificatori C; il testo da rimpiazzare è arbitrario. Generalmente

il testo di sostituzione è rappresentato dal resto della linea; una lunga definizione può continuare sulle linee successive inserendo un \ alla fine della linea. La "visibilità" di un nome definito con **#define** va dal proprio punto di definizione alla fine del file sorgente. I nomi possono essere ridefiniti ed una definizione può usare precedenti definizioni. All'interno di stringhe tra apici non vengono effettuate sostituzioni per cui, ad esempio, se **YES** è un nome definito, in **printf("YES")** non verrà effettuata alcuna sostituzione.
Poiché l'implementazione di **#define** non fa parte del compilatore vero e proprio ma è un passaggio a priori riguardante le macro, ci sono molte poche limitazioni grammaticali su ciò che può essere definito. Per esempio, i patiti dell'Algol potrebbero definire

```
#define then
#define begin {
#define end   ;}
```

e poi scrivere

```
if (i > 0) then
    begin
        a = 1;
        b = 2;
    end
```

È inoltre possibile definire macro con argomenti cosicché il testo di sostituzione dipende dal modo con cui la macro viene chiamata. Come esempio, definiamo una macro chiamata **max** in questo modo

```
#define max(A, B)   ((A) > (B) ? (A) : (B))
```

Ora la linea

```
x = max(p+q, r+s);
```

verrà rimpiazzata dalla linea

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Ciò fornisce una "funzione massimo" che espande in codice vero e proprio invece che in una chiamata di funzione. Se gli argomenti vengono trattati coerentemente, questa macro servirà per ogni tipo di dati; non c'è bisogno di definire vari tipi di **max** a seconda dei vari tipi di dati, come bisognerebbe invece fare con le funzioni.
Naturalmente, esaminando l'espansione di **max** qui sopra, si noteranno alcune trappole. Le espressioni vengono valutate due volte; questo è un inconveniente se ciò causa effetti collaterali tipo chiamate di funzione ed operatori di incremento. Bisogna adottare alcuni accorgimenti con le parentesi per assicurarsi che venga mantenuto l'ordine di valutazione. (Si consideri la macro

```
#define square(x)  x * x
```

quando viene chiamata con **square(z+1)**). Si presentano anche alcuni problemi propriamente lessicali: non dev'esserci alcuno spazio tra il nome della macro e la parentesi sinistra che introduce il suo elenco di argomenti.
Comunque, le macro sono preziose. Un esempio pratico viene dato dalla libreria standard di **I/O** che verrà trattata nel Capitolo 7, in cui **getchar** e **putchar** sono definite come macro (naturalmente **putchar** vuole un argomento) per evitare il costo di una chiamata di funzione per ogni carattere.
Nell'Appendice A vengono descritte altre caratteristiche del preprocessore di macro.

Esercizio 4-9. Definire una macro **swap(x, y)** che scambia i suoi due argomenti **int**. (La struttura a blocchi può essere d'aiuto).

CAPITOLO 5

PUNTATORI ED ARRAY

Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile. I puntatori sono molto usati in C, in parte perché rappresentano talvolta l'unica maniera per esprimere un calcolo, ed in parte perché generalmente portano ad un codice più compatto ed efficiente di quanto possa essere ottenuto con altri procedimenti.
I puntatori, assieme all'istruzione **goto**, sono stati considerati una maniera meravigliosa per creare programmi impossibili da capire. Questo è certamente vero quando vengono usati senza un preciso motivo, e diventa facile creare puntatori che puntano dove non ci si aspetterebbe. Con criterio, comunque, i puntatori possono essere usati per ottenere chiarezza e semplicità. Questo è l'aspetto che tenteremo di illustrare.

5.1 Puntatori e Indirizzi

Poiché un puntatore contiene l'indirizzo di un oggetto, è possibile accedere all'oggetto "indirettamente" tramite il puntatore. Supponiamo che **x** sia una variabile, per esempio un **int**, e che **px** sia un puntatore, creato in maniera non ancora specificata. L'operatore unario **&** dà l'*indirizzo* di un oggetto, così l'istruzione

```
px = &x;
```

assegna l'indirizzo di **x** alla variabile **px**; si dice ora che **px** "punta a **x**". L'operatore **&** può essere applicato solamente alle variabili ed agli elementi degli array; costrutti del tipo di **&(x+1)** e **&3** non sono ammessi. Non è neanche permesso ottenere l'indirizzo di una variabile **register**.
L'operatore unario * tratta il suo operando come l'indirizzo dell'obiettivo finale ed accede all'indirizzo per prenderne il contenuto, per cui se anche **y** è un **int**,

```
y = *px;
```

assegna ad **y** il contenuto di quálunque cosa a cui punti **px**. Così la sequenza

```
px = &x;
y = *px
```

assegna lo stesso valore ad **y** come

```
y = x;
```

È inoltre necessario dichiarare le variabili che partecipano a tutto ciò:

```
int x, y;
int *px;
```

La dichiarazione di **x** e di **y** non sono nuove, mentre lo è la dichiarazione del puntatore **px**.

```
int *px;
```

è inteso come mnemonico; comunica che la combinazione ***px** è un **int**, per cui, se **px** compare nel contesto ***px**, è equivalente ad una variabile di tipo **int**. In effetti, la sintassi della dichiarazione di una variabile assomiglia alla sintassi delle espressioni in cui la variabile può comparire. Questo fato è utile in tutti i casi in cui si ha a che fare con dichiarazioni complicate. Per esempio

```
double atof(), *dp;
```

comunica che, in un'espressione, **atof()** e ***dp** hanno valori di tipo **double**.
Bisogna anche notare che la dichiarazione di un puntatore implica che debba puntare ad un particolare tipo di oggetto.
I puntatori possono comparire nelle espressioni. Per esempio, se **px** punta all'intero **x**, allora ***px** può apparire in ogni contesto in cui può apparire **x**.

```
y = *px + 1
```

assegna ad **y** uno più di **x**;

```
printf("%d\n", *px)
```

stampa il valore corrente di **x**; e

```
d = sqrt((double) *px)
```

produce in **d** la radice quadrata di **x**, che viene forzata in un **double** prima di entrare in **sqrt** (vedi Capitolo 2).
In espressioni del tipo

```
y = *px + 1
```

gli operatori unari * ed **&** hanno priorità maggiore degli operatori aritmetici, per cui questa espressione prende ciò che viene puntato da **px**, gli aggiunge uno e lo assegna ad **y**. Vedremo fra breve cosa può voler dire

```
y = *(px + 1)
```

Riferimenti a puntatori possono anche apparire nella parte sinistra di un assegnamento. Se **px** punta ad **x**, allora

```
*px = 0
```

assegna zero ad **x** e

```
*px += 1
```

lo incrementa, come anche

```
(*px)++
```

Nell'ultimo esempio le parentesi sono necessarie; senza di esse, l'espressione incrementerebbe **px** al posto dell'oggetto a cui punta, poiché gli operatori unari come * e ++ vengono valutati da destra a sinistra.
Infine, poiché i puntatori sono variabili, possono essere manipolati come le altre variabili. Se **py** è un altro puntatore ad **int**, allora

```
py = px
```

copia il contenuto di **px** all'interno di **py**, facendo in modo che **py** punti a ciò a cui punta **px**.

5.2 Puntatori e Argomenti di Funzioni

Siccome il C passa gli argomenti alle funzioni tramite "chiamata per valore", per la funzione non esiste un modo diretto per alterare una variabile della funzione chiamante. Cosa bisogna fare se occorre proprio cambiare un argomento? Per esempio, una routine di ordinamento potrebbe scambiare due elementi non in ordine attraverso una funzione chiamata **swap**. È sufficiente scrivere

```
swap(a, b);
```

in cui la funzione **swap** è definita come

```
swap(x, y)          /* SBAGLIATA */
int x, y;
{
     int temp;

     temp = x;
     x = y;
     y = temp;
}
```

Con la chiamata per valore, **swap** *non può* modificare gli argomenti **a** e **b** nella routine con cui è stata chiamata.
Fortunatamente, esiste un modo per ottenere l'effetto desiderato. Il programma chiamante passa i *puntatori* ai valori che devono essere modificati:

```
swap(&a, &b);
```

Poiché l'operatore **&** riporta l'indirizzo di una variabile, **&a** è un puntatore ad **a**. All'interno di **swap** stessa, gli argomenti vengono dichiarati come puntatori e si ha accesso agli operandi attraverso di essi.

```
swap(px, py)          /* scambia *px e *py */
int *px, *py;
{
     int temp;

     temp = *px;
     *px = *py;
     *py = temp;
}
```

Un uso comune degli argomenti puntatori è nelle funzioni che devono ritornare più di un singolo valore (si potrebbe dire che **swap** ritorna due valori, i nuovi valori dei suoi argomenti). Come esempio, consideriamo una funzione **getint** che opera una conversione da un input in formato libero spezzando una serie di caratteri in valori interi, un intero per ogni chiamata. **getint** deve ritornare il valore che ha trovato oppure un segnale di fine file quando è terminato l'input. Questi valori devono ritornare come oggetti separati, senza preoccuparsi del valore che viene usato per **EOF**, che potrebbe anche essere il valore di un intero in input.
Una soluzione basata sulla funzione di input **scanf**, che descriveremo nel Capitolo 7, consiste nel fatto che **getint** ritorni **EOF** come proprio valore di funzione, se si trova la fine del file; ogni altro valore di ritorno segnala un normale intero. Il valore numerico dell'intero trovato viene ritornato attraverso l'argomento che deve essere un puntatore ad un intero. Questa organizzazione separa lo stato di fine file dai valori numerici.
Il ciclo seguente riempie un array con interi attraverso le chiamate a **getint**:

```
int n, v, array[AMPIEZZA];

for (n = 0; n < AMPIEZZA && getint(&v) != EOF; n++)
     array[n] = v;
```

Ogni chiamata assegna a **v** il successivo intero trovato nell'input. Da notare che è fondamentale scrivere **&v** al posto di **v** come argomento di **getint**. Usando semplicemente **v** è facile causare un errore di indirizzamento, poiché **getint** è convinta di avere a disposizione un puntatore valido.
getint stessa è una chiara modifica della già scritta **atoi**:

```
getint(pn)     /* prende il successivo intero dall'input */
int *pn;
{
    int c, segno;

    while ((c = getch()) == ' ' || c == '\n' || c == '\t')
        ;       /* salta gli spazi bianchi */
    segno = 1;
    if (c == '+' || c == '-')  {     /* segno */
        segno = (c=='+') ? 1 : -1;
        c = getch();
    }
    for (*pn = 0; c >= '0' && c <='9'; c = getch())
        *pn = 10 * *pn + c - '0';
    *pn *= segno;
    if (c != EOF)
        ungetch(c);
    return(c)
}
```

In tutta **getint, *pn** viene usato come una semplice variabile **int**. Abbiamo poi usato **getch** e **ungetch** (descritti nel Capitolo 4), così il carattere in più che deve essere letto può essere rimandato nell'input.

Esercizio 5-1. Scrivere **getfloat**, l'analoga di **getint** in floating point. Che tipo deve ritornare **getfloat** come proprio valore di funzione?

5.3 Puntatori ed Array

In C esiste una stretta relazione tra puntatori ed array, talmente stretta che dovrebbero essere oggetto di una trattazione unica. Ogni operazione che può essere ottenuta dall'indicizzazione di un array, può anche essere ottenuta con i puntatori. La versione con i puntatori in genere è più veloce, ma almeno all'inizio più difficile da comprendere. La dichiarazione

```
int a[10]
```

definisce un array **a** di lunghezza 10, cioè un blocco di 10 oggetti consecutivi chiamati **a[0], a[1], ..., a[9].** La notazione **a[i]** significa l'elemento dell'array nella posizione **i**-esima a partire dall'inizio. Se **pa** è un puntatore ad un intero, dichiarato come

```
int *pa
```

allora l'assegnamento

```
pa = &a[0]
```

fa sì che **pa** punti all'elemento zero di **a**; cioè **pa** contiene l'indirizzo di **a[0]**. Ora l'assegnamento

```
x = *pa
```

copia il contenuto di **a[0]** dentro **x**.
Se **pa** punta ad un particolare elemento di un array **a**, allora *per definizione* **pa+1** punta al successivo elemento, e più in generale **pa-i** punta ad **i** elementi prima di **pa** e **pa+i** punta ad **i** elementi dopo. Allora, se **pa** punta ad **a[0]**,

```
*(pa+1)
```

si riferisce al contenuto di **a[1], pa+i** è l'indirizzo di **a[i]**, e ***(pa+i)** è il contenuto di **a[i]**.
Queste osservazioni sono valide indipendentemente dal tipo delle variabili dell'array **a**.
La definizione di "aggiungere uno ad un puntatore" e, per estensione, tutta l'aritmetica dei puntatori, consiste nel fatto che l'incremento è proporzionato alla occupazione di memoria dell'oggetto a cui punta. Allora in **pa+i**, **i** viene moltiplicato per la lunghezza dell'oggetto a cui punta **pa** prima di essere sommato a **pa**.
La corrispondenza tra indici ed aritmetica dei puntatori è chiaramente molto stretta. Infatti un riferimento in un array viene convertito dal compilatore ad un puntatore all'inizio dell'array. Il risultato è che il nome di array *è* un'espressione di tipo puntatore. Ciò ha alcune utili implicazioni. Siccome il nome di un array è un sinonimo per la locazione dell'elemento zero, l'assegnamento

```
pa = &a[0]
```

può anche essere scritto come

```
pa = a
```

Ancora più sorprendente, almeno a prima vista, è il fatto che un riferimento ad **a[i]** può anche essere scritto come ***(a+i)**. Nel valutare **a[i]** il C lo converte immediatamente in ***(a+i)**; le due forme sono perfettamente equivalenti. Applicando l'operatore **&** ad entrambe le parti di questa equivalenza, ne consegue che **&a[i]** e **a+i** sono pure identici: **a+i** è l'indirizzo dell'i-esimo elemento dopo **a**. Come altra faccia della medaglia, se **pa** è un puntatore, le espressioni possono utilizzarlo con un indice: **pa[i]** è identico a ***(pa+i)**. In breve, ogni espressione array ed indice può essere scritta come un puntatore ed un valore di scarto e viceversa, anche nella stessa istruzione.
C'è una differenza di cui bisogna tener conto tra il nome di un array ed un puntatore. Un puntatore è una variabile, per cui **pa=a** e **pa++** sono valide operazioni. Ma il nome di un array è una *costante*, non una variabile: non sono accettate costruzioni come: **a=pa** oppure **a++** oppure **p=&a**.
Quando il nome di un array viene passato ad una funzione, in realtà viene passata la locazione dell'inizio dell'array. All'interno di una funzione, questo argomento è una variabile allo stesso modo di ogni altra variabile, per cui un nome di array come argomento è veramente un puntatore, cioè una variabile contenente un indirizzo.

Possiamo utilizzare questo fatto per scrivere una nuova versione di **strlen**, che calcola la lunghezza di una stringa.

```
strlen(s)     /* ritorna la lunghezza della stringa s */
char *s;
{
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return(n);
}
```

L'incremento di **s** è permesso, poiché è una variabile di tipo puntatore; **s++** non ha alcun effetto sulla stringa di caratteri nella funzione che ha chiamato **strlen**, ma incrementa semplicemente la copia privata dell'indirizzo di **strlen**.
Come parametri formali in una definizione di funzione,

```
char s[];
```

e

```
char *s;
```

sono esattamente equivalenti; la scelta viene largamente determinata da come saranno scritte le operazioni all'interno della funzione. Quando viene passato il nome di un array ad una funzione, la funzione può assumere a seconda di come conviene, di aver ricevuto o un array o un puntatore, manipolandolo in accordo alla scelta. Se appare appropriato e chiaro, può anche usare entrambi i tipi di operazioni.
È possibile passare una parte di un array ad una funzione, passando il puntatore all'inizio del sottoarray. Per esempio, se **a** è un array,

```
f(&a[2])
```

e

```
f(a+2)
```

passano entrambe alla funzione **f** l'indirizzo dell'elemento **a[2]**, poiché **&a[2]** ed **a+2** sono entrambe espressioni di puntatori che si riferiscono al terzo elemento di **a**. In **f,** la dichiarazione dell'argomento può essere

```
f(arr)
int arr[];
{

}
```

oppure

```
f(arr)
int *arr;
{

}
```

Per quanto riguarda **f**, il fatto che l'argomento si riferisce ad una parte di un array più esteso, non ha alcuna conseguenza.

5.4 Aritmetica sugli Indirizzi

Se **p** è un puntatore, allora **p++** incrementa **p** in modo tale che punti al successivo elemento di qualsiasi tipo di oggetto a cui punta **p**, e **p+=i** incrementa **p** in maniera che punta **i** elementi successivi a quello a cui puntava. Queste e altre simili costruzioni sono le forme più semplici e comuni di aritmetica sui puntatori o indirizzi.
Nel suo approccio all'aritmetica degli indirizzi, il C è compatto e regolare; l'integrazione di puntatori, array ed aritmetica degli indirizzi rappresenta un punto di forza del linguaggio. Vediamo alcune delle sue proprietà scrivendo un rudimentale allocatore di memoria (comunque utile rapportato alla sua semplicità). Ci sono due routines: **alloc(n)** ritorna un puntatore **p** ad **n** caratteri consecutivi, che possono essere usati dal chiamante di **alloc** per memorizzare caratteri; **free(p)** rilascia la memoria così acquisita per un successivo riutilizzo. Le routines sono "rudimentali" poiché le chiamate a **free** devono comparire in ordine opposto alle chiamate ad **alloc** per cui la memoria gestita da **alloc** e **free** è uno stack o lista del tipo ultimo-arrivato, primo-uscito. La libreria standard del C fornisce funzioni analoghe che non hanno alcun tipo di restrizione; nel Capitolo 8 vedremo la versione migliorata. Nel frattempo, comunque, molte applicazioni hanno solamente bisogno di una grezza **alloc** per fornire un certo spazio in memoria ad un istante imprevedibile.
La più semplice implementazione è che **alloc** distribuisca pezzi di un grosso array di caratteri che chiameremo **allocbuf**. Questo array è privato ad **alloc** e **free**. Poiché trattano puntatori, non indici di array, nessun'altra routine ha bisogno di sapere il nome dell'array che può così essere dichiarato esterno **static**, cioè locale al file sorgente contenente **alloc** e **free**, invisibile dall'esterno. Nelle implementazioni reali, l'array potrebbe anche essere privo di nome; potrebbe essere ottenuto richiedendo al sistema operativo un puntatore a qualche blocco di memoria senza nome.
Un'altra informazione di cui si ha bisogno è quanto è stato usato di **allocbuf**. Viene usato un puntatore al successivo elemento libero, chiamato **allocp**. Quando **alloc** viene chiamata per avere **n** caratteri, verifica se è rimasto sufficiente spazio in **allocbuf**. Se c'è, **alloc** ritorna il valore corrente di **allocp** (ad esempio, l'inizio del blocco libero) e lo incrementa di **n** puntando alla successiva area libera. **free(p)** assegna semplicemente **p** ad **allocp** se **p** è all'interno di **allocbuf**.

```
#define  NULL 0     /* valore del puntatore che segnala errore */
#define  ALLOC_AMPIO 1000 /* ampiezza dello spazio disponibile */

static char allocbuf[ALLOC_AMPIO];     /* memoria per alloc */
static char *allocp = allocbuf      /* successiva posizione libera */

char *alloc(n)  /* ritorna un puntatore ad n caratteri */
int n;
{
    if (allocp + n <= allocbuf + ALLOC_AMPIO)  { /* da' spazio */
        allocp += n;
        return(allocp -n);  /* vecchio p */
    } else          /* non c'e' abbastanza spazio */
        return(NULL)
}

free(p)         /* libera la memoria puntata da p */
char *p;
{
    if (p >= allocbuf && p < allocbuf + ALLOC_AMPIO)
        allocp = p;
}
```

In generale un puntatore può essere inizializzato come ogni altra variabile, sebbene normalmente gli unici valori sensati sono **NULL** (discusso nel seguito) o un'espressione che coinvolge indirizzi di dati del tipo appropriato precedentemente definito. La dichiarazione

```
static char *allocp = allocbuf;
```

definisce **allocp** come puntatore di carattere e lo inizializza a puntare ad **allocbuf**, che è la successiva posizione libera al momento dell'inizio del programma. Ciò potrebbe anche essere scritto come:

```
static char *allocp = &allocbuf[0];
```

poiché il nome dell'array è l'indirizzo dell'elemento zero; entrambe le forme sono accettabili.
Il test

```
if (p >= allocbuf && p < allocbuf + ALLOC_AMPIO)
```

verifica se c'è abbastanza spazio per soddisfare una richiesta di **n** caratteri. Se c'è, il nuovo valore di **allocp** può al massimo oltrepassare di uno la fine di **allocbuf**. Se le richieste possono essere soddisfatte, **alloc** ritorna un normale puntatore (si noti la dichiarazione della funzione stessa). Altrimenti, **alloc** deve ritornare un segnale che non c'è più spazio. Il C garantisce che nessun particolare che legittimamente punta a dati contenga zero, così un valore di ritorno uguale a zero può essere utilizzato per segnalare un evento anormale, in questo caso che non c'è spazio.

Abbiamo scritto **NULL** al posto di zero per indicare più chiaramente che è un valore speciale per i puntatori. In generale, non ha senso assegnare un intero ad un puntatore; zero è un caso speciale.
Test come

```
if (allocp + n <= allocbuf + ALLOC_AMPIO)
```

e

```
if (allocp + n <= allocbuf + ALLOC_AMPIO)
```

mostrano parecchie importanti caratteristiche dell'aritmetica dei puntatori. Per prima, in certe circostanze i puntatori possono essere confrontati. Se **p** e **q** puntano ad elementi dello stesso array, le relazioni del tipo <, >=, ecc., lavorano correttamente.

```
p < q
```

è vera, per esempio, se **p** punta ad un elemento precedente a quello a cui punta **q** nell'array. Inoltre funzionano le relazioni == e !=. Ogni puntatore può essere confrontato in uguaglianza o diseguaglianza con **NULL** però ogni vantaggio viene meno se si effettuano operazioni aritmetiche o confronti tra puntatori che puntano ad array differenti. Se si è fortunati, si avranno risultati assurdi su tutte le macchine. Se si è sfortunati, il programma lavorerà correttamente su una macchina ma crollerà misteriosamente su un'altra.
Abbiamo già osservato che un puntatore ed un intero possono essere sommati o sottratti. La costruzione

```
p + n
```

significa l'ennesimo oggetto oltre quello a cui correntemente punta **p**. Ciò è vero indipendentemente dal tipo di oggetto a cui è stato dichiarato che **p** punta; il compilatore proporziona **n** tenendo conto della dimensione dell'elemento a cui **p** punta, che viene determinato dalla dichiarazione di **p**. Per esempio, sul **PDP/11**, i fattori di scala sono 1 per **char**, 2 per **int** e **short**, 4 per **long** e **float** ed 8 per **double**.
È inoltre valida la sottrazione di puntatori: se **p** e **q** puntano ad elementi dello stesso array, **p-q** è il numero di elementi tra **p e q**. Questo fatto può essere utilizzato per scrivere un'altra versione di **strlen**:

```
strlen(s)     /* ritorna la lunghezza della stringa s */
char *s;
{
    char *p = s;

    while (*p != '\0')
        p++;
    return(p-s);
}
```

Nelle sue dichiarazioni, **p** viene inizializzato ad **s**, cioè a puntare al primo carattere.

Nel ciclo **while**, di volta in volta viene esaminato ogni carattere finché si incontra **\0** alla fine. Poiché **\0** è zero, e poiché **while** verifica solamente se l'espressione è zero, è possibile omettere il test esplicito, e cicli del genere vengono scritti come

```
while (*p)
    p++;
```

Poiché **p** punta a caratteri, **p++** fa avanzare ogni volta **p** al carattere successivo, e **p-s** riporta il numero di caratteri scanditi, cioè la lunghezza della stringa. L'aritmetica dei puntatori è coerente: se si stava trattando con dei **float** che occupano più memoria dei **char**, e se **p** fosse stato un puntatore a **float, p++** sarebbe avanzato al successivo **float**. Così possiamo scrivere un'altra versione di **alloc** che gestisce, per esempio, dei **float** al posto dei **char**, cambiando solamente **char** con **float** in tutta **alloc** e **free**. Tutte le operazioni sui puntatori tengono automaticamente contro della grandezza dell'oggetto puntato, in maniera tale che non vada modificato nient'altro.
Oltre alle operazioni qui menzionate (sommare e sottrarre un puntatore ed un intero; sottrarre o confrontare due puntatori), qualsiasi altro tipo di aritmetica coi puntatori non è permessa. Non è possibile sommare due puntatori o moltiplicarli o dividerli od operare su di essi con degli shift o mascherarli o aggiungere ad essi **float** o **double**.

5.5 Puntatori a caratteri e Funzioni

Una *stringa costante*, scritta come

```
"Sono una stringa"
```

è un array di caratteri. Nella rappresentazione interna, il compilatore termina l'array con il carattere **\0** in maniera che il programma possa trovarne la fine. La lunghezza in memoria è così uno in più del numero di caratteri contenuti tra i doppi apici.
Forse l'utilizzo più comune delle stringhe costanti è come argomenti alle funzioni, come in

```
printf("salve, mondo\n");
```

Quando in un programma appare una stringa di caratteri come questa, l'accesso ad essa viene ottenuto attraverso un puntatore a carattere; ciò che **printf** riceve è un puntatore all'array di caratteri.
Gli array di caratteri naturalmente non hanno bisogno di essere argomenti di funzioni. Se **messaggio** è dichiarato come

```
char *messaggio
```

allora l'istruzione

```
messaggio = "ci vediamo domani sera";
```

assegna a **messaggio** un puntatore ai caratteri indicati. Questo *non* è copiare una

stringa; vengono chiamati in causa solamente i puntatori. Il C non possiede alcun operatore che tratti un'intera stringa come un'unità.
Vedremo altri aspetti dei puntatori e degli array studiando due utili funzioni della libreria standard di **I/O** che verrà trattata nel Capitolo 7.
La prima funzione è **strcpy(s, t)** che copia la stringa **t** sulla stringa **s**. Gli argomenti sono scritti in questo ordine per analogia all'assegnamento, in cui si scriverebbe

```
s = t
```

per assegnare **t** ad **s**. La prima versione è con gli array:

```
strcpy(s, t)     /* copia t in s */
char s[], t[];
{
    int i;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

Per confronto, questa è una versione di **strcpy** con i puntatori.

```
strcpy(s, t)     /* copia t in s; versione con i puntatori 1 */
char *s, *t;
{
    while ((*s = *t) != '\0')  {
        s++;
        t++;
    }
}
```

Poiché gli argomenti vengono passati per valore, **strcpy** può usare **s** e **t** liberamente. Qui ci sono puntatori inizializzati convenientemente che vengono fatti avanzare nell'array un carattere per volta, finché viene copiato in **s** il **0** che termina **t**.
In pratica, **strcpy** non sarebbe scritta come abbiamo mostrato qui sopra. Una seconda possibilità potrebbe essere

```
strcpy(s, t)     /* copia t in s; versione con i puntatori 2 */
char *s *t;
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

In quest'ultima l'incremento di **s** e **t** è spostato nella parte di test. Il valore di ***t**++ è il carattere a cui **t** puntava prima che **t** fosse incrementato; il postfisso ++ non modifica **t** fino a che il carattere non è stato analizzato. Allo stesso modo, il carattere viene collocato nella vecchia posizione di **s** prima che **s** venga incrementato. Questo carattere è anche il valore che viene confrontato con **0** per controllare il ciclo. L'effetto finale è che i caratteri vengono copiati da **t** ad **s** fino alla fine includendo il **0** finale.

Osserviamo anche che il confronto con **0** risulta ridondante, per cui la funzione viene spesso scritta come

```
strcpy(s, t)    /* copia t in s; versione con i puntatori 3 */
char *s, *t;
{
    while (*s++ = *t++)
        ;
}
```

Sebbene a prima vista può apparire criptico, la convenienza di notazione è considerevole; questo modello dev'essere oggetto di attenzione, se non altro perché lo vedrete frequentemente in programmi C.
La seconda routine è **strcmp(s, t)** che confronta le stringhe di caratteri **s** e **t** e ritorna un numero negativo, zero o un numero positivo se **s** è alfabeticamente minore, uguale o maggiore di **t**. Il valore di ritorno viene ottenuto sottraendo i caratteri alla prima posizione in cui **s** e **t** divergono.

```
strcmp (s, t)    /* ritorna <0 se s<t, 0 se s==t, >0 se s>t */
char s[], t[];
{
    int i;

    i = 0;
    while (s[i] == t[i])
        if (s[i++] == '\0')
            return(0);
    return(s[i] - t[i]);
}
```

La versione coi puntatori di **strcmp**:

```
strcmp (s, t)    /* ritorna <0 se s<t, 0 se s==t, >0 se s>t */
char *s, *t;
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return(0);
    return(*s - *t);
}
```

Poiché + + e – – sono operatori sia prefissi che postfissi, possono apparire atre combinazioni di * e + + e – –, anche se meno frequentemente. Per esempio,

```
*++p
```

incrementa **p** *prima* di prendere in considerazione il carattere a cui punta **p**.

```
*--p
```

decrementa prima **p**.

Esercizio 5-2. Scrivere una versione con i puntatori della funzione **strcat** che abbiamo mostrato nel Capitolo 2: **strcat(s, t)** copia la stringa **t** alla fine di **s**.

Esercizio 5-3. Scrivere una macro per **strcpy**.

Esercizio 5-4. Riscrivere i programmi dei precedenti capitoli esercitandosi con i puntatori al posto degli indici di array. Buone possibilità sono offerte da **getline** (Capitoli 1 e 4), **atoi, itoa** e le loro varianti (Capitoli 2, 3 e 4), **reverse** (Capitolo 3), **index** e **getop** (Capitolo 4).

5.6 I Puntatori non sono Interi

Potreste notare nei vecchi programmi C una certa disinvoltura riguardo alla copia di puntatori. È generalmente vero che in molte macchine un puntatore può essere assegnato ad un intero e viceversa senza cambiamenti; non viene fatta alcuna conversione e non vengono persi bit. Purtroppo, ciò ha portato ad una certa presa di libertà con routines che ritornano puntatori che vengono poi semplicemente passati ad altre routines: vengono spesso tralasciate le dichiarazioni dei puntatori necessari. Per esempio consideriamo la funzione **strsave(s)**, che copia la stringa **s** salvandola in qualche posto, ottenuta da una chiamata ad **alloc**, e ritorna un puntatore ad esso. In maniera corretta, dovrebbe essere scritta come

```
char *strsave(s)     /* salva la stringa s da qualche parte */
char *s;
{
    char *p, *alloc();

    if ((p = alloc(strlen(s)+1)) != NULL)
        strcpy(p, s);
    return(p);
}
```

In pratica, ci sarebbe la tendenza ad omettere le dichiarazioni:

```
strsave(s)     /* salva la stringa s da qualche parte */
char *s;
{
    char *p;

    if ((p = alloc(strlen(s)+1)) != NULL)
        strcpy(p, s);
    return(p);
}
```

Questo funzionerà su molte macchine, perché il tipo di default per gli argomenti delle funzioni è **int**, e generalmente **int** e puntatori possono essere assegnati gli uni agli altri. Tuttavia questo tipo di codice porta a qualche rischio poiché dipende da dettagli di implementazione e dall'architettura della macchina che potrebbero non essere presi in

considerazione dal particolare compilatore che si sta usando. È meglio essere completi in tutte le dichiarazioni. (Il programma *lint* metterà in guardia circa queste costruzioni nel caso che si utilizzino inavvertitamente).

5.7 Array Multi-dimensionali

Il C supporta array a più dimensioni, sebbene in pratica si tenda ad usarli molto meno degli array di puntatori. In questo paragrafo, vedremo alcune delle loro proprietà.
Consideriamo il problema di convertire date dal giorno del mese al giorno dell'anno e viceversa. Per esempio, il primo Marzo è il 60esimo giorno di un anno non bisestile ed il 61esimo di un anno bisestile. Definiamo due funzioni per ottenere la conversione: **giorno_anno** converte il mese e giorno nel giorno dell'anno, e **mese_giorno** converte il giorno dell'anno nel mese e giorno. Poiché quest'ultima funzione ritorna due valori, gli argomenti mese e giorno saranno puntatori:

```
mese_giorno(1977, 60, &m, &g)
```

assegna 3 ad **m** e 1 a **g** (Marzo 1).
Entrambe le funzioni necessitano delle medesime informazioni: una tabella del numero di giorni per ogni mese ("trenta giorni ha Novembre...").
Poiché il numero di giorni per mese cambia a seconda che l'anno sia o meni bisestile, è più semplice separarli in due linee in un array a due dimensioni piuttosto di analizzare durante i calcoli cosa succede a Febbraio. L'array e le funzioni per ottenere le trasformazioni sono i seguenti:

```
static int gior_mese[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

giorno_anno(anno, mese, giorno)     /* determina il giorno dell'anno */
int anno, mese, giorno;             /* dal mese e giorno */
{
    int i, bisest;

    bisest = anno%4 == 0 && anno%100 != 0 || anno%400 == 0;
    for (i = 1; i < mese; i++)
        giorno += gior_mese[bisest][i];
    return(giorno);
}

mese_giorno(anno, annogior, pmese, pgiorno)  /* determina mese e giorno *
int anno, annogior, *pmese, *pgiorno;        /* dal giorno dell'anno */
{
    int i, bisest;

    bisest = anno%4 == 0 && anno%100 != 0 || anno%400 == 0;
    for (i = 1; annogior > gior_mese[bisest][i]; i++)
        annogior -= gior_mese[bisest][i];
    *pmese = i;
    *pgiorno = annogior;
}
```

L'array **gior_mese** dev'essere esterno ad entrambe **giorno_anno** e **mese_giorno** in modo che possano entrambe utilizzarlo.
gior_mese è il primo array a due dimensioni con cui abbiamo a che fare. In C, per definizione un array a due dimensioni è in verità un array ad una dimensione in cui ognuno dei suoi elementi è un array. Quindi gli indici vengono scritti come

```
gior_mese[i][j]
```

invece che

```
gior_mese[i, j]
```

come in molti linguaggi. A parte questo, un array a due dimensioni può essere trattato in linea di massima nello stesso modo degli altri linguaggi. Gli elementi vengono assegnati in memoria per righe, cioè l'indice più a destra varia più velocemente per il fato che accede agli elementi nell'ordine di memorizzazione.
Un array viene inizializzato da una lista di valori di inizializzazione tra graffe; ogni riga di un array a due dimensioni viene inizializzata da una corrispondente sottolista. Abbiamo cominciato l'array **gior_mese** con una colonna di zero così i numeri dei mesi vanno naturalmente da 1 a 12 invece che da 0 a 11. Poiché lo spazio qui non è un fattore limitante, è più comodo che adattare gli indici dell'array.
Se un array a due dimensioni viene passato ad una funzione, la dichiarazione dell'argomento nella funzione *deve* includere la dimensione della colonna; la dimensione della riga è irrilevante poiché ciò che viene passato è, come prima, un puntatore. In questo particolare caso, è un puntatore ad oggetti che sono array di 13 **int**. Per cui se l'array **gior_mese** dev'essere passato alla funzione **f,** la dichiarazione di **f** sarà:

```
f(gior_mese)
int gior_mese[2][13];
{

}
```

La dichiarazione dell'argomento di **f** può anche essere

```
int gior_mese[][13];
```

poiché il numero di righe è irrilevante, o potrebbe essere

```
int (*gior_mese)[13];
```

affermando che l'argomento è un puntatore ad un array di 13 interi. Le parentesi sono necessarie perché le quadre **[]** hanno priorità più alta di *; senza parentesi, la dichiarazione

```
int *gior_mese[13];
```

è un array di 13 puntatori ad interi, come vedremo nel prossimo paragrafo.

5.8 Array di Puntatori; Puntatori a Puntatori

Poiché i puntatori sono essi stessi variabili ci si potrebbe aspettare un uso degli array di puntatori. Difatti è così. Vediamoli scrivendo un programma che ordina un insieme di linee di testo in ordine alfabetico, una semplice versione del programma di utilità **UNIX** *sort*.
Nel Capitolo 3 abbiamo presentato una funzione di Shell sort che ordinava un array di interi. Usiamo il medesimo algoritmo con la differenza che ora si ha a che fare con linee di testo di diversa lunghezza e che diversamente dagli interi, non possono essere confrontate e spostate con una singola operazione. Si ha bisogno di una struttura dati che faccia fronte alle linee di testo di lunghezza variabile in maniera efficiente.
È qui che entrano in causa gli array di puntatori. Se le linee da ordinare vengono assegnate con le estremità che si toccano in un lungo array di caratteri (magari mantenuto da **alloc**), allora si può avere accesso ad ogni linea con un puntatore al suo primo carattere.
I puntatori stessi possono essere contenuti in un array. Due linee possono essere confrontate passando i loro puntatori a **strcmp**. Quando devono essere scambiate due linee non in ordine, vengono scambiati i *puntatori* nell'array di puntatori, non le linee di testo stesse. Ciò elimina il doppio problema di una complicata gestione di memoria e delle pesanti conseguenze che si avrebbero spostando le linee.
Il processo di ordinamento richiede 3 passi:

leggi tutte le linee dell'input
ordinale
stampale in ordine

Come al solito è meglio dividere il programma in funzioni che si adattino a questa naturale divisione, con una routine principale che le controlla.
Per il momento lasciamo perdere la fase di ordinamento e concentriamoci sulla struttura dei dati e sull'input e output. La routine di input deve accumulare e salvare i caratteri di ogni linea, e costruire un array di puntatori alle linee. Inoltre deve contare il numero delle linee di input poiché questa informazione serve per ordinarle e per stamparle. Poiché la funzione di input può avere a che fare solo con un numero finito di linee di input, se si riceve troppo input, può ritornare qualche valore non permesso di conteggio linee, come −1. La routine di output deve solo stampare le linee nell'ordine in cui appaiono nell'array di puntatori.

```
#define NULL 0
#define LINEE 100 /* massime linee da ordinare */

main()   /* ordina le linee di input */
{
    char *lineptr[LINEE];  /* puntatori alle linee di testo */
    int  nlinee;      /* numero di linee lette dall'input */

    if ((nlinee = readlines(lineptr, LINEE)) >= 0)  {
        sort(lineptr, nlinee);
        writelines(lineptr, nlinee);
    }
    else
        printf("input troppo grande da ordinare\n");
}
```

```
#define MAXLUNG 1000

readlines(lineptr, maxlinee)    /* legge linee di input */
char *lineptr[];                /* per l'ordinamento */
int maxlinee;
{
    int lung, nlinee;
    char *p, *alloc(), linea[MAXLUNG];

    nlinee = 0;
    while ((lung = getline(linea, MAXLUNG)) > 0)
        if (nlinee >= maxlinee)
            return(-1);
        else if ((p = alloc(lung)) == NULL)
            return(-1);
        else {
            linea[lung-1] = '\0';  /* toglie il newline */
            strcpy(p, linea);
            lineptr[nlinee++] = p;
        }
    return(nlinee);
}
```

Il newline alla fine di ogni linea viene tolto in maniera che non abbia effetto nel criterio in cui vengono ordinate le linee.

```
writelines(lineptr, nlinee)    /* scrive le linee di output */
char *lineptr[];
int nlinee;
{
    int i;

    for (i = 0; i < nlinee; i++)
        printf("%s\n",lineptr[i]);
}
```

La novità principale è la dichiarazione di **lineptr**:

```
char *lineptr[LINEE];
```

afferma che **lineptr** è un array di **LINEE** elementi, ogni elemento del quale è un puntatore a **char**. Cioè, **lineptr[i]** è un puntatore di carattere e ***lineptr[i]** accede al carattere.
Poiché **lineptr** è esso stesso un array che viene passato a **writelines**, può essere trattato come un puntatore esattamente nella stessa maniera vista negli ultimi esempi, e la funzione può essere scritta anche come

```
writelines(lineptr, nlinee)    /* scrive le linee di output */
char *lineptr[];
int nlinee;
{
    while (--nlinee >= 0)
        printf("%s\n", *lineptr++);
}
```

***lineptr** punta inizialmente alla prima linea; ogni incremento lo avanza alla successiva linea mentre **nlinee** viene decrementata.
Con l'input e l'output sotto controllo, possiamo procedere nell'ordinamento. Lo Shell sort del Capitolo 3 necessita di pochi cambiamenti: le dichiarazioni devono essere modificate e l'operazione di confronto dev'essere spostata in una funzione separata. L'algoritmo di base rimane lo stesso, che ci assicura il corretto funzionamento.

```
sort(v, n)      /* ordina le stringhe v[0] ... v[n-1] */
char *v[];      /* in ordine crescente */
int n;
{
    int salto, i, j;
    char *temp;

    for (salto = n/2; salto > 0; salto /= 2)
        for (i = salto; i < n; i++)
            for (j = i-salto; i >= 0; j-=salto) {
                if (strcmp(v[j], v[j+salto) <= 0)
                    break;
                temp = v[j];
                v[j] = v[j+salto];
                v[j+salto] = temp;
            }
}
```

Poiché ogni singolo elemento di **v** (alias **lineptr**) è un puntatore a carattere, per poterne copiare uno sull'altro, deve esistere anche **temp**.
Abbiamo scritto il programma il più linearmente possibile per farlo funzionare nel minor tempo possibile. Potrebbe essere più veloce, ad esempio, copiare le linee entranti direttamente in un array gestito da **readlines**, piuttosto che copiarle all'interno di **linea** e poi in un luogo non visibile mantenuto da **alloc**. Ma come prima scrittura del programma è più saggio fare in modo che sia di semplice comprensione, preoccupandosi dell'"efficienza" più tardi. Per fare in modo che questo programma sia significativamente più veloce, è inutile evitare una copia non necessaria delle linee di input; è più adeguato rimpiazzare lo Shell sort con qualcosa di meglio, come ad esempio un Quicksort, per notare una differenza.
Nel Capitolo 1 abbiamo fatto notare che siccome i cicli **while** e **for** verificano la condizione di termine almeno una volta *prima* di eseguire il corpo del ciclo, contribuiscono ad assicurare che il programma funzioni nei propri limiti, in particolare senza input.
È interessante passare attraverso le funzioni del programma di ordinamento per verificare cosa succede quando non c'è alcun testo di input.

Esercizio 5-5. Riscrivere **readlines** creando le linee in un array fornito da **main** invece che chiamare **alloc** per gestire la memoria. Di quanto risulta più veloce il programma?

5.9 Inizializzazione degli array di Puntatori

Consideriamo il problema di scrivere la funzione **nome_mese(n)** che ritorna un puntatore alla stringa di caratteri contenente il nome dell'**n-esimo** mese. Questa è una applicazione ideale per un array interno **static. nome_mese** contiene un array privato

di stringhe di caratteri, e quando viene chiamata ritorna un puntatore alla stringa adeguata. L'argomento di questo paragrafo è come viene inizializzato l'array di nomi. La sintassi è molto simile alle precedenti inizializzazioni:

```
char *nome_mese(n)  /* ritorna il nome del n-esimo mese */
int n;
{
    static char *nome[] ={
        "mese scorretto",
        "Gennaio",
        "Febbraio",
        "Marzo",
        "Aprile",
        "Maggio",
        "Giugno",
        "Luglio",
        "Agosto",
        "Settembre",
        "Ottobre",
        "Novembre",
        "Dicembre"
    };

    return((n < 1 || n > 12) ? nome[0] : nome[n]);
}
```

La dichiarazione di **nome**, che è un array di puntatori a carattere, è uguale a quella di **lineptr** nell'esempio di ordinamento. L'inizializzazione è semplicemente una lista di stringhe di caratteri; ognuna viene assegnata alla corrispondente posizione nell'array. Più precisamente, i caratteri dell'**i-esima** stringa vengono piazzati da qualche altra parte, e in **nome[i]** viene messo un puntatore ad essa. Poiché non è specificata la grandezza dell'array **nome**, il compilatore stesso conta le inizializzazioni e ne determina il numero corretto.

5.10 Confronto tra Puntatori e Array Multi-dimensionali

Chi inizia col C non ha le idee chiare circa la differenza tra un array a due dimensioni ed un array di puntatori, come nome nell'esempio precedente. Dando la dichiarazione

```
int a[10][10];
int *b[10];
```

l'uso di **a** e **b** può essere simile, nel fatto che **a[5][5]** e **b[5][5]** sono entrambi validi riferimenti ad un singolo **int**. Ma **a** è un array vero e proprio: sono state allocate tutte le 100 celle di memoria, e per trovare ogni dato elemento viene usato il convenzionale calcolo di indici. Per **b** invece la dichiarazione alloca solamente 10 puntatori; ognuno dev'essere assegnato a puntare ad un array di interi. Assumendo che ognuno punti ad un array di 10 elementi, verranno riservate 100 celle di memoria più le 10 celle per i puntatori. In questo modo l'array di puntatori usa un po' di più di memoria e può richiedere un'esplicita fase di inizializzazione. Ha però che vantagi: l'accesso ad un elemento viene effettuato con l'indirizzamento indiretto attraverso un puntatore invece

che da una moltiplicazione e un'addizione, e le righe dell'array possono essere di lunghezza diversa. Cioè, ogni elemento di **b** non deve per forza puntare ad un array di 10 elementi; alcuni potrebbero puntare a due elementi, altri a venti ed altri ancora a nessuno.
Benché abbiamo sostenuto questa discussione in termini di interi, l'uso di gran lunga più frequente degli array di puntatori è come mostrato in **nome_mese**: per memorizzare stringhe di caratteri di diversa lunghezza.

Esercizio 5-6. Riscrivere la routine **giorno_anno** e **mese_giorno** con i puntatori invece delle indicizzazioni.

5.11 Argomenti sulla linea di comando

Negli ambienti che supportano il C, esiste un modo per passare argomenti sulla linea di comando o parametri ad un programma nel momento in cui inizia l'esecuzione. Quando viene chiamata **main** per iniziare l'esecuzione, viene chiamata con due argomenti. Il primo (per convenzione chiamato **argc**) è il numero di argomenti presenti nella linea di comando con cui è stato chiamato il programma; il secondo (**argv**) è un puntatore ad un array di stringhe di caratteri che contiene gli argomenti, uno per stringa. Manipolare queste stringhe di caratteri rappresenta un uso comune di livelli multipli di puntatori.
L'esempio più semplice dell'uso e delle dichiarazioni necessarie è il programma **echo**, che dà una semplice eco su una singola linea degli argomenti presenti sulla propria linea di comando, separati da spazi. Cioè, se viene dato il comando

```
echo salve, mondo
```

l'output è

```
salve, mondo
```

Per convenzione, **argv[0]** è il nome con cui è stato chiamato il programma, così **argc** è perlomeno 1. Nell'esempio precedente, **argc** è 3 e **argv[0], argv[1]** e **argv[2]** sono rispettivamente "**echo**," "**salve**", e "**mondo**". Il primo argomento vero e proprio è **argv[1]** e l'ultimo è **argv[argc−1]**. Se **argc** è 1, non ci sono argomenti sulla linea di comando dopo il nome del programma. Ciò si può vedere anche in **echo**:

```
main(argc, argv)    /* fa una eco degli argomenti; la versione */
int argc;
char *argv[];
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

Poiché **argv** è un puntatore ad un array di puntatori, vi sono parecchi motivi per scrivere

questo programma in modo che richiami la manipolazione dei puntatori invece che l'indicizzazione di un array. Vediamo due varianti.

```
main(argc, argv)    /* fa una eco degli argomenti; 2a versione */
int argc;
char *argv[];
{
    while (--argc > 0)
        printf("%s%c", *++argv, (argc > 1) ? ' ' : '\n');
}
```

Poiché **argv** è un puntatore all'inizio dell'array delle stringhe di argomento, incrementarlo di 1 **(++argv)** fa in modo che punti all'originale **argv[1]** al posto di **argv[0]**.
Ogni successivo incremento lo sposta di volta in volta sul prossimo argomento; ***argv** è così il puntatore a tale argomento. Nello stezzo momento, viene decrementato **argc**; quando diventa zero, non rimangono più argomenti da stampare.
Alternativamente,

```
main(argc, argv)    /* fa una eco degli argomenti; 3a versione */
int argc;
char *argv[];
{
    while (--argc > 0)
        printf((argc > 1) ? "%s " : "%s\n", *++argv);
}
```

Questa versione mostra che il formato dell'argomento di **printf** può essere un'espressione tale e quale ognuna delle altre. Questo uso non è molto frequente, ma vale la pena ricordarlo.
Come secondo esempio, facciamo qualche miglioramento al programma di ricerca pattern del Capitolo 4. Se vi ricordate, abbiamo incorporato il pattern di ricerca profondamente nel programma, una soluzione ovviamente insoddisfacente. Seguendo la strada della utility **UNIX** *grep*, cambiamo il programma in maniera che il modello da soddisfare venga specificato dal primo argomento nella linea di comando.

```
#define MAXLINEE 1000

main(argc, argv)  /* cerca il pattern dal primo argomento */
int argc;
char *argv[];
{
    char linea[MAXLINEE];

    if (argc != 2)
        printf("Uso: find pattern\n");
    else
        while(getline(linea, MAXLINEE) > 0)
            if (index(linea, argv[1] >= 0)
                printf("%s", linea);
}
```

Si può ora elaborare il modello base per illustrare ulteriori costruzioni con puntatori. Supponiamo di voler permettere l'uso di due argomenti opzionali. Uno significa "stampa

tutte le linee *eccetto* quelle che incontrano il pattern", il secondo "precedi ogni linea stampata con il suo numero di linea".
Una convenzione per i programmi C è che un argomento che inizia con il segno meno, introduce un segnale o parametro opzionale. Se si sceglie **−x** (per "except") per segnalare il capovolgimento, e **−n** ("number") per richiedere la numerazione delle linee, allora il comando

```
find -x -n the
```

con l'input

```
When she gets there she knows
if the stores are all closed
with a word she can get
what she came for
```

deve produrre l'output

```
3: with a word she can get
4: what she came for
```

Gli argomenti opzionali devono essere permessi in qualsiasi ordine, ed il resto del programma dovrebbe essere indipendente dal numero degli argomenti che erano in realtà presenti. In particolare, la chiamata ad **index** non dovrebbe riferirsi ad **argv[2]** quando c'è un singolo argomento e ad **argv[1]** quando non ce ne sono.
Inoltre, per l'utente è più comodo che gli argomenti opzionali possano essere concatenati, come in

```
find -nx the
```

Questo è il programma

```
#define MAXLINEA 1000

main(argc, argv)  /* cerca il pattern dal primo argomento */
int argc;
char *argv[];
{
    char linea[MAXLINEA], *s;
    long lineanum = 0;
    int except = 0, number = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        for (s = argv[0]+1; *s != '\0'; s++)
            switch (*s) {
            case 'x':
                except = 1;
                break;
            case 'n':
                number = 1;
                break;
            default:
                printf("find: opzione scorretta %c\n", *s);
                argc = 0;
                break;
            }
    if (argc != 1)
        printf("Uso: find -x -n pattern\n");
    else
        while (getline(linea, MAXLINEA) > 0) {
            lineanum++;
            if ((index(linea, *argv) >= 0) != except) {
                if (number)
                    printf("%ld: ", lineanum);
                printf("%s", linea);
            }
        }
}
```

argv viene incrementato prima di ogni argomento opzionale, e **argc** viene decrementato. Se non ci sono errori, alla fine del ciclo **argc** dovrebbe essere 1 e ***argv** dovrebbe puntare al pattern. Da notare che ***++argv** è un puntatore ad una stringa di argomento; **(*++argv)[0]** è il suo primo carattere. Le parentesi sono necessarie poiché in assenza di esse l'espressione sarebbe ***++(argv[0])**, che è piuttosto differente (e sbagliata). Una valida forma alternativa potrebbe essere ***++argv**.

Esercizio 5-7. Scrivere il programma **add** che valuta un'espressione in notazione Polacca inversa dalla linea di comando. Per esempio

```
add 2 3 4 + *
```

valuta 2 x (3+4).

Esercizio 5-8. Modificare i programmi **entab** e **detab** (scritti come esercizio nel Capitolo 1) facendo in modo che accetti una lista di tab come argomenti. Se non ci sono argomenti, usare il normale valore del tab.

Esercizio 5-9. Estendere **entab** e **detab** in modo che accetti l'abbreviazione.

```
entab m +n
```

indicando in *n* colonne l'ampiezza del tab, partendo dalla colonna *m*. Scegliere un funzionamento di default conveniente (per l'utente).

Esercizio 5-10. Scrivere il programma **tail**, che stampa le ultime *n* linee del proprio input. Per default, *n* è 10, ma può essere cambiato da un argomento opzionale, così

```
tail -n
```

stampa le ultime *n* linee. Il programma dovrebbe comportarsi razionalmente senza problemi anche per input o valori di *n* assurdi. Si scriva il programma in maniera che faccia il miglior uso della memoria disponibile: le linee devono essere memorizzate come in **sort**, non in un array a due dimensioni di ampiezza prefissata.

5.12 Puntatori a Funzioni

Nel C, una funzione non è in sé una variabile, ma è possibile definire *un puntatore ad una funzione*, che può essere manipolato, passato a funzioni, inserito in un array e così via. Vedremo questo modificando la procedura di ordinamento scritta poco fa in questo capitolo in modo che se viene dato l'argomento opzionale **−n**, ordinerà le linee di input numericamente invece che alfabeticamente.
Un ordinamento consiste spesso di tre parti — un *confronto* che determina l'ordinamento di ogni coppia di oggetti, uno *scambio* che inverte il loro ordine, ed un *algoritmo di ordinamento* che effettua confronti e scambi finché gli oggetti non sono in ordine. L'algoritmo di ordinamento è indipendente dalle operazioni di confronto e scambio, per cui passandogli differenti funzioni di confronto e scambio, possiamo ottenere degli ordinamenti per differenti criteri. Questo è l'approccio che abbiamo dato al nuovo ordinamento.
Il confronto alfabetico di 2 linee viene ottenuto da **strcmp** e lo scambio da **swap** come prima; abbiamo inoltre bisogno di una routine **numcmp** che confronta due linee sulla base dei valori numerici e ritorna lo stesso tipo di indicatore della condizione di **strcmp**.

Queste tre funzioni vengono richiamate in **main** ed a **sort** vengono passati i puntatori ad esse. **sort** chiama di volta in volta le funzioni attraverso i puntatori. Abbiamo evitato la condizione di errore per gli argomenti per concentrarci sulla questione principale.

```
#define LINEE 100 /* massime linee da ordinare */

main(argc,argv)     /* ordina le linee di input */
int argc;
char *argv[];
{
    char *lineptr[LINEE]; /* puntatori alle linee di testo */
    int nlinee;         /* numero di linee di input lette */
    int strcmp(), numcmp();  /* funzioni di confronto */
    int swap();     /* funzione di scambio */
    int numerico = 0;   /* 1 se ordinamento numerico */

    if (argc>1 && argv[1][0] == '-' && argv[1][1] == '\n')
        numerico = 1;
    if ((nlinee = readlines(lineptr, LINEE)) >= 0) {
        if (numerico)
            sort(lineptr, nlinee, numcmp, swap);
        else
            sort(lineptr, nlinee, strcmp, swap);
        writelines(lineptr, nlinee);
    } else
        printf("input troppo grande da ordinare\n");
}
```

strcmp, numcmp e **swap** sono indirizzi di funzioni; poiché sono conosciute come funzioni, non è necessario l'operatore **&**, nello stesso modo in cui non è necessario prima del nome di un array. Il compilatore predispone affinché venga passato l'indirizzo della funzione.
Il secondo passo è modificare **sort**:

```
sort(v, n, comp, exch)   /* ordina le stringhe v[0]...v[n-1] */
char *v[];               /* in ordine crescente */
int n;
int (*comp)(), (*exch)();
{
    int salto, i, j;

    for (salto = n/2; salto > 0; salto /= 2)
        for (i = salto; i < n; i++)
            for (j = i-salto; j >= 0; j-= salto) {
                if ((*comp)(v[j], v[j+salto]) <= 0)
                    break;
                (*exch)(&v[j], &v[j+salto]);
            }
}
```

Le dichiarazioni devono essere oggetto di particolare attenzione.

```
int (*comp)()
```

afferma che **comp** è un puntatore ad una funzione che ritorna un **int**. Il primo set di parentesi è necessario; senza di esse,

```
int *comp()
```

affermerebbe che **comp** è una funzione che ritorna un puntatore ad un **int**, che è una cosa assai diversa.
L'uso di **comp** nella linea

```
if ((*comp)(v[j], v[j+salto]) <= 0)
```

è conforme con la dichiarazione: **comp** è un puntatore ad una funzione, ***comp** è la funzione, e

```
(*comp)(v[j], v[j+salto])
```

è la sua chiamata. Le parentesi sono richieste affinché i componenti vengano correttamente associati.
Abbiamo già visto **strcmp**, che confronta due stringhe. Questo è **numcmp**, che confronta due stringhe sulla base di un valore numerico:

```
numcmp(s1, s2) /* confronta s1 ed s2 numericamente */
char *s1, *s2;
{
    double atof(), v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return(-1);
    else if (v1 > v2)
        return(1);
    else
        return(0);
}
```

L'ultimo passo è quello di aggiungere la funzione **swap** che scambia due puntatori. Essa viene modificata direttamente da ciò che abbiamo presentato precedentemente in questo capitolo.

```
swap(px , py)  /* scambia *px e *py */
char *px[], *py[];
{
    char *temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

C'è una varietà di altre opzioni che possono essere aggiunte al programma di ordinamento; alcune di esse costituiscono stimolanti esercizi.

Esercizio 5-11. Modificare **sort** in modo che accetti l'opzione **−r**, che indica un ordinamento inverso (decrescente). Naturalmente **−r** deve poter lavorare con **−n**.

Esercizio 5-12. Aggiungere l'opzione **−f** che racchiude insieme maiuscole e minuscole, in modo che nell'ordinamento non si faccia distinzione tra maiuscole e minuscole: lettere maiuscole e minuscole vengono ordinate insieme, affinché **a** ed **A** appaiano adiacenti e non separate da un'intera sequenza di alfabeto.

Esercizio 5-13. Aggiungere l'opzione **−d** ("ordine di dizionario"), che confronta solamente lettere, numeri e spazi. Assicurarsi che lavori insieme a **−f**.

Esercizio 5-14. Aggiungere la possibilità di lavorare per campi, in modo che l'ordinamento possa essere svolto su campi all'interno di linee, ogni campo in conformità ad un set indipendente di opzioni. (L'indice di questo libro è stato ordinato con **−df** per i termini dell'indice e **−n** per i numeri delle pagine).

CAPITOLO 6

STRUTTURE

Una *struttura* è un insieme di una o più variabili, eventualmente di tipo diverso, raggruppate sotto un unico nome per consentirne un trattamento più comodo. (Le strutture sono chiamate "records" in alcuni linguaggi, tra i quali ricordiamo il Pascal).
Un tipico esempio di struttura è la registrazione delle paghe: un impiegato viene descritto da un insieme di attributi come il nome, l'indirizzo, il numero di previdenza sociale, lo stipendio, ecc. Alcuni di essi potrebbero a loro volta essere strutture: un nome ha diverse componenti, come ne hanno un indirizzo ed anche uno stipendio.
Le strutture aiutano ad organizzare dati complessi, particolarmente in grossi programmi, poiché in molte situazioni permettono di trattare un gruppo di variabili in relazione tra di loro come un'unità e non come entità separate. In questo capitolo cercheremo di illustrare come vengono usate le strutture. I programmi che useremo sono più lunghi di molti altri nel libro, ma rimagono di dimensioni modeste.

6.1 Fondamenti

Rivediamo le routines di conversione della data del Capitolo 5. Una data consiste di diverse parti, come il giorno, il mese e l'anno, e magari anche il giorno dell'anno e il nome del mese. Queste cinque variabili possono essere disposte in un'unica struttura come segue:

```
struct data {
    int giorno;
    int mese;
    int anno;
    int giornoanno;
    char nome_mese[4];
};
```

La parola chiave **struct** introduce una dichiarazione di struttura, che è una lista di dichiarazioni tra parentesi graffe. Un nome opzionale chiamato *structure tag* può seguire la parola struct (come **data** in questo esempio). Questo nome identifica questo tipo di struttura, e può essere in seguito usato come abbreviazione in luogo della dichiarazione dettagliata.
Gli elementi o variabili nominati in una struttura sono chiamati membri. Un membro o il nome di una struttura e una variabile normale (ad esempio che non sia un membro) possono avere lo stesso nome senza conflitto, in quanto possono essere sempre distinti

dal contesto. Naturalmente, per ragioni di stile, si dovrebbe usare lo stesso nome solo per oggetti intimamente collegati.
La parentesi graffa chiusa che termina la lista dei membri può essere seguita da una lista di variabili, come un qualunque altro nome di tipo basilare. Cioè

```
struct { ... } x, y, z;
```

è sintatticamente analogo a

```
int x, y, z;
```

nel senso che entrambe le istruzioni dichiarano che **x**, **y**, e **z** sono variabili del tipo menzionato e fanno in modo che per esse sia allocato spazio in memoria.
Una dichiarazione di struttura che non sia seguita da una lista di variabili non occupa memoria; descrive semplicemente la *forma* della struttura. Se la dichiarazione è etichettata, in ogni caso, l'etichetta può essere usata successivamente in definizioni di istanze reali della struttura. Per esempio, con riferimento alla dichiarazione di struttura **data** di cui sopra,

```
struct data d;
```

definisce una variabile **d** come struttura di tipo **data**. Una struttura esterna o statica può essere inizializzata seguendone la definizione con una lista di valori iniziali per i suoi componenti:

```
struct data d ={ 25, 4, 1945, 115, "Apr" };
```

In un'espressione qualsiasi si può fare riferimento ad un membro di una particolare struttura con un costrutto della forma

nome-della-struttura.membro

L'operatore di membro di struttura "." associa il nome della struttura al nome del membro. Per ricavare la variabile **bisest** utilizzando la struttura **d**, per esempio,

```
bisest = ( d.anno % 4 == 0 && d.anno % 100 != 0 || d.anno % 400 == 0 );
```

o, per controllare il nome del mese,

```
if (strcmp(d.nome_mese, "Ago") == 0) ...
```

o, per convertire il primo carattere del mese in minuscolo,

```
d.nome_mese[0] = lower(d.nome_mese[0]);
```

Le strutture possono essere nidificate; un record paghe potrebbe di fatto essere:

```
struct persona {
    char nome[DIMNOME];
    char indirizzo[DIMIND];
    long cap;
    long cod_fisc;
    double salario;
    struct data nascita;
    struct data assunzione;
};
```

La struttura **persona** contiene due date. Se si dichiara **imp** come

```
struct persona imp;
```

allora

```
imp.nascita.mese
```

si riferisce al mese di nascita. L'operatore membro di struttura "." associa da sinistra a destra.

6.2 Strutture e funzioni

Ci sono un certo numero di restrizioni sulle strutture C. Le regole essenziali sono che le sole operazioni che si possono eseguire su di una struttura sono prendere il suo indirizzo con **&**, e accedere a uno dei suoi membri. Questo implica che alle strutture non si può assegnare o non si possono copiare come un tutt'uno, e che non possono essere passate a funzioni o ritornate da esse. (Queste restrizioni saranno eliminate in versioni future). In ogni caso, i puntatori alle strutture non soffrono di queste limitazioni e così strutture e funzioni lavorano insieme comodamente. Infine, le strutture automatiche, come gli array dello stesso tipo, non possono essere inizializzate; possono esserlo solo quelle esterne o statiche.
Approfondiamo alcuni di questi punti riscrivendo le funzioni di conversione della data del precedente capitolo usando le strutture. Siccome le regole vietano di passare una struttura ad una funzione direttamente, dovremo passarne o i singoli componenti direttamente, o il puntatore all'intera struttura. La prima alternativa usa **giorno_anno** come scritto nel Capitolo 5:

```
d.giornoanno = giorno_anno(d.anno,d.mese, d.giorno);
```

L'altro modo consiste nel passare un puntatore. Se abbiamo dichiarato **assunzione** come

```
struct data assunzione;
```

e riscritto **giorno_anno**, allora possiamo dire

```
assunzione.giornoanno = giorno_anno(&assunzione);
```

per passare a **giorno_anno** un puntatore ad **assunzione**. La funzione deve essere modificata poiché il suo argomento ora è un puntatore invece di una lista di variabili.

```
giorno_anno(pd)     /* assegna il giorno dell'anno da mese e giono */
struct data *pd;
{
    int i, giorno, bisest;

    giorno = pd->anno;
    bisest = pd->anno % 4 == 0 && pd->anno % 100 != 0
                || pd->anno % 400 == 0;
    for (i = 1; i < pd->mese; i++)
        giorno += gior_mese[bisest][i];
    return(giorno);
}
```

La dichiarazione

```
struct data *pd;
```

afferma che **pd** è un puntatore ad una struttura di tipo **data**. La notazione esemplificata da

```
pd->anno
```

è nuova. Se **p** è un puntatore ad una struttura, allora

p−> *membro-della-struttura*

si riferisce a quel membro in particolare. (L'operatore −> è un segno meno seguito da un >.)
Siccome **pd** punta ad una struttura, ci si potrebbe riferire al membro **anno** anche con

```
(*pd).anno
```

ma i puntatori a strutture sono usati così frequentemente che la notazione −> è fornita come una utile abbreviazione. Le parentesi in **(*pd).anno** sono necessarie poiché la presenza dell'operatore di membro di struttura **.** è più elevata di *. Entrambi −> e **.** associano da sinistra a destra, per cui

```
p->q->memb
imp.nascita.mese
```

stanno per

```
(p->q)->memb
(imp.nascita).mese
```

Per completezza riportiamo l'altra funzione, **mese__giorno**, riscritta usando la struttura.

```
mese_giorno(pd)     /* assegna mese e giorno dal giorno dell'anno */
struct data *pd;
{
    int i, bisest;

    bisest = pd->anno % 4 == 0 && pd->anno % 100 != 0
             || pd->anno % 400 == 0;
    pd->giorno = pd->giornoanno;
    for (i = 1; pd->giorno > gior_mese[bisest][i]; i++)
        pd->giorno -= gior_mese[bisest][i];
    pd->mese = i;
}
```

Gli operatori per strutture −> e ., insieme alle **()** delle liste di argomenti e **[ ]** degli indici, sono al culmine della gerarchia delle precedenze e perciò legano strettamente tra di loro. Per esempio, data la dichiarazione

```
struct {
    int x;
    int *y;
} *p;
```

allora

```
++p->x
```

incrementa **x**, non **p**, perché ciò equivale implicitamente a ++**(p−>x)**. Le parentesi possono essere usate per alterare la precedenza: **(++p)−>x** incrementa **p** prima di accedere ad **x**, e **(p++)−>x** incrementa **p** dopo. (Quest'ultima coppia di parentesi non è necessaria. Perché?)
Allo stesso modo, ∗**p**−>y accede all'oggetto puntato da **y** qualunque esso sia; ∗**p−>y**++ incrementa **y** dopo avere accesso a ciò a cui punta (come ∗**s**++); **(∗p−>y)**++ incrementa l'oggetto puntato da **y**; e ∗**p++−>y** incrementa **p** dopo avere accesso all'oggetto puntato da **y**.

6.3 Arrays di strutture

Le strutture sono particolarmente adatte per gestire arrays di variabili in relazione tra loro. Per esempio, considerate un programma che conti la frequenza delle parole chiave del C. Abbiamo bisogno di un array di stringhe di caratteri per contenere i nomi, e un array di interi per i contatori. Una possibilità è quella di usare due array paralleli **p__chiave** e **conta__par**, come in

```
char *p_chiave[NCHIAVI];
int  conta_par[NCHIAVI];
```

Ma il fatto stesso che i due arrays siano paralleli indica che è possibile un organizzazione differente. Ogni indicazione di parola chiave è in realtà una coppia:

```
char *p_chiave;
int conta_par;
```

ed ecco un array di coppie. La dichiarazione di struttura

```
struct chiave {
    char *p_chiave;
    int  conta_par;
} tabchiavi[NCHIAVI];
```

definisce un array **tabchiavi** di strutture di questo tipo, e alloca memoria per esse. Ciascun elemento dell'array è una struttura. Lo stesso potrebbe essere scritto anche

```
struct chiave {
    char *p_chiave;
    int  conta_par;
};

struct chiave tabchiavi[NCHIAVI];
```

E poiché la struttura **tabchiavi** contiene in realtà un insieme costante di nomi, è molto semplice inizializzarla una volta per tutte quando la si definisce. L'inizializzazione di una struttura è analoga a quelle precedenti: la definizione è seguita da una lista di inizializzatori tra parentesi graffe:

```
struct chiave {
    char *p_chiave;
    int  conta_par;
} tabchiavi[] ={
    "break", 0,
    "case", 0,
    "char", 0,
    "continue", 0,
    "default", 0,
    /* ... */
    "unsigned", 0,
    "while", 0
};
```

I valori iniziali sono elencati in coppie corrispondenti ai membri della struttura. Sarebbe più preciso racchiudere i valori iniziali di ciascuna "riga" o struttura tra parentesi graffe, come in

```
{"break", 0},
{"case", 0},
...
```

ma le parentesi più interne non sono necessarie quando i valori iniziali sono variabili semplici o stringhe di caratteri, e quando sono tutti presenti. Come al solito, il compilatore calcolerà il numero di elementi nell'array **tabchiavi** se i valori iniziali sono presenti e le **[ ]** sono lasciate vuote.

Il programma di conteggio delle parole chiave inizia con la definizione di **tabchiavi**. La routine principale legge l'input chiamando ripetutamente una funzione chiamata **getword** che accetta l'input ad una parola per volta. Ogni parola viene cercata in **tabchiavi** con una versione della funzione di ricerca binaria scritta nel Capitolo 3. (Naturalmente la lista delle parole chiave deve essere fornita in ordine crescente perché tutto questo funzioni.)

```
#define    MAXPAROLA    20

main()    /* conta le parole chiave del C */
{
    int n, t;
    char parola[MAXPAROLA];

    while ((t = getword(parola, MAXPAROLA)) != EOF)
        if (t == LETTERA)
            if ((n = binaria(parola, tabchiavi, NCHIAVI)) >= 0)
                tabchiavi[n].conta_par++;
    for (n = 0; n < NCHIAVI; n++)
        if (tabchiavi[n].conta_par > 0)
            printf("%4d %s\n",
                tabchiavi[n].conta_par, tabchiavi[n].p_chiave);
}

binaria(parola, tab, n)  /* trova parola in tab[0]...tab[n-1] */
char *parola;
struct chiave tab[];
int n;
{
    int basso, alto, mezzo, cond;

    basso = 0;
    alto = n - 1;
    while (basso <= alto) {
        mezzo = (basso + alto) /2;
        if ((cond = strcmp(parola, tab[mezzo].p_chiave)) < 0)
            alto = mezzo -1;
        else if (cond > 0)
            basso = mezzo +1;
        else
            return(mezzo);
    }
    return(-1);
}
```

Mostreremo tra poco la funzione **getword**: per ora basta dire che ritorna **LETTERA** ogni volta che trova una parola, e copia la parola nel suo primo argomento.
La quantità **NCHIAVI** è il numero di parole chiave in **tabchiavi**. Anche se potremmo contarlo a mano, è di gran lunga più semplice e più sicuro farlo con la macchina, specialmente se la lista è suscettibile di cambiamenti. Una possibilità sarebbe quella di terminare la lista dei valori iniziali con un puntatore nullo, poi ciclare lungo **tabchiavi** fino a trovarne la fine.
Ma questo è più del necessario, perché la dimensione dell'array è determinata completamente in fase di compilazione. Il numero di elementi non è altro che

dimensione di **tabchiavi** / *dimensione di* **struct chiave**

Il C fornisce un operatore unario che agisce in fase di compilazione chiamato **sizeof** che può essere usato per calcolare le dimensioni di un oggetto qualsiasi. L'espressione

sizeof *(oggetto)*

produce un intero uguale alla dimensione dell'oggetto specificato. (La dimensione è data in una unità non meglio specificata chiamata "byte", che equivale in dimensioni a un oggetto **char**.) L'oggetto di sizeof può essere una variabile vera e propria, o un array, o il nome di un tipo base come **int** o **double**, o il nome di un tipo derivato come una struttura. Nel nostro caso, il numero di parole chiave è la dimensione dell'array diviso per la dimensione di un elemento dell'array. Questo calcolo può essere usato in una istruzione #**define** per determinare il valore di **NCHIAVI**:

```
#define    NCHIAVI    (sizeof(tabchiavi) / sizeof(struct chiave))
```

Ed ora passiamo alla funzione **getword**. In realtà abbiamo scritto una **getword** più generale di quanto serva per questo programma, ma effettivamente non è molto più complicata. **getword** ritorna la prossima "parola" dell'input, dove una parola è una stringa di lettere e cifre che inizia con una lettera, oppure un singolo carattere. Il tipo di oggetto viene restituito sotto forma di valore di funzione; è **LETTERA** se il simbolo è una parola, **EOF** per fine file o il carattere stesso se non è alfabetico.

```
getword(w, lim)          /* prendi la prossima parola dall'input */
char *w;
int lim;
{
    int c, t;

    if (tipo(c = *w++ = getch()) != LETTERA) {
        *w = '\0';
        return(c);
    }
    while (--lim > 0) {
        t = tipo(c = *w++ = getch());
        if (t != LETTERA && t != CIFRA) {
            ungetch(c);
            break;
        }
    }
    *(w-1) = '\0';
    return(LETTERA);
}
```

getword usa le routines **getch** e **ungetch** scritte nel capitolo 4: quando termina la scansione di un simbolo alfabetico, **getword** è andata un carattere in là di troppo. La chiamata ad **ungetch** respinge indietro il carattere per la prossima chiamata.
getword chiama **tipo** per determinare il tipo di ogni carattere singolo dell'input. Ecco una versione esclusivamente per l'alfabeto ASCII.

```
tipo(c)         /* ritorna il tipo di un carattere ASCII */
int c;
{
    if ( c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
        return(LETTERA);
    else if (c >= '0' && c <= '9')
        return(CIFRA);
    else
        return(c);
}
```

Le costanti simboliche **LETTERA** e **CIFRA** possono avere qualsiasi valore che non sia in conflitto con i caratteri non alfanumerici ed **EOF**; le ovvie scelte sono

```
#define LETTERA  'a'
#define CIFRA    '0'
```

getword può essere più veloce se le chiamate alla funzione **tipo** sono sostituite da accessi ad un appropriato array **tipo[]**. La libreria standard del C fornisce macro chiamate **isalpha** ed **isdigit** che operano in questo modo.

Esercizio 6-1. Apportare quest'ultima modifica a **getword** e misurare il cambiamento di velocità del programma.

Esercizio 6-2. Scrivere una versione di **tipo** che sia indipendente dal set di caratteri.

Esercizio 6-3. Scrivere una versione del programma che conta le parole chiave che non conti le occorrenze contenute in stringhe tra doppi apici.

6.4 Puntatori alle strutture

Per illustrare alcune delle considerazioni che scaturiscono dai puntatori e dagli array di strutture, riscriviamo il programma che conta le parole chiave usando stavolta i puntatori invece degli indici di array.
La dichiarazione esterna di **tabchiavi** non necessita di cambiamenti, ma **main** e **binaria** hanno bisogno di modifiche.

```
main()    /* conta le parole chiave del C; versione con puntatori */
{
    int t;
    char parola[MAXPAROLA];
    struct chiave *binaria(), *p;

    while ((t = getword(parola, MAXPAROLA)) != EOF)
        if (t == LETTERA)
            if ((p=binaria(parola, tabchiavi, NCHIAVI)) != NULL)
                p->conta_par++;
    for (p = tabchiavi; p < tabchiavi + NCHIAVI; p++)
        if (p->conta_par > 0)
            printf("%4d %s\n",p->conta_par, p->p_chiave);
}
```

```
struct chiave *binaria(parola, tab, n)     /* trova la parola */
char *parola;                              /* in tab[0]...tab[n-1] */
struct chiave tab[];
int n;
{
    int cond;                                  struct alb_nodo *sinistra;
    struct chiave *basso = &tab[0];
    struct chiave *alto = &tab[n-1];
    struct chiave *mezzo;

    while (basso <= alto) {
        mezzo = basso + (alto-basso) / 2;
        if ((cond = strcmp(parola, mezzo->p_chiave)) < 0)
            alto = mezzo - 1;
        else if (cond > 0)
            basso = mezzo + 1;
        else
            return(mezzo);
    }
    return(NULL);
}
```

Ci sono parecchie cose degne di nota. Innanzitutto, la dichiarazione di **binaria** deve indicare che il valore di ritorno è un puntatore al tipo struttura **chiave**, invece che un intero; ciò è dichiarato sia in **main** sia in **binaria**. Se **binaria** trova la parola, ritorna un puntatore a questa; se fallisce, ritorna **NULL**.
In secondo luogo, tutto l'accesso agli elementi di **tabchiavi** è realizzato tramite puntatori. Questo comporta un cambiamento significativo a **binaria**: il calcolo dell'elemento mediano non può più essere semplicemente

```
mezzo = (basso+alto) / 2
```

perché l'*addizione* di due puntatori non produrrà alcun valore utile (anche quando è divisa per 2), e infatti non è permessa. Deve essere cambiato in

```
mezzo = basso + (alto - basso) / 2
```

che fa sì che **mezzo** punti all'elemento a metà strada tra **basso** e **alto**.
Dovreste anche analizzare i valori iniziali di **basso** e **alto**. È possibile inizializzare un puntatore con l'indirizzo di un oggetto definito precedentemente; esattamente ciò che abbiamo fatto qui.
In **main** abbiamo scritto

```
for (p = tabchiavi; p < tabchiavi + NCHIAVI; p++)
```

Se **p** è un puntatore ad una struttura, ogni aritmetica su **p** tiene conto della reale dimensione della struttura, cosicché **p**++ incrementa **p** della quantità esatta per accedere al prossimo elemento dell'array di strutture. Ma non crediate che la dimensione di una struttura sia la somma delle dimensioni dei propri membri: a causa delle condizioni di allineamento di oggetti diversi ci possono essere "buchi" nella struttura.

Infine, una parentesi sull'aspetto del programma. Quando una funzione ritorna un tipo complicato, come in

```
struct chiave *binaria(parola, tab, n)
```

il nome della funzione può essere difficile da vedere, e da trovare con un text editor. Di conseguenza, a volte si usa uno stile alternativo:

```
struct chiave *
binaria(parola, tab, n)
```

Questo è per lo più questione di gusto personale; decidete quale delle forme preferite ed usatela sempre.

6.5 Strutture ricorsive

Supponiamo di voler trattare il problema più generale che consiste nel contare la frequenza con cui ricorrono *tutte* le parole in qualche input. Siccome non si conosce la lista di parole in anticipo, non la si può ordinare e usare una ricerca binaria. In più non possiamo effettuare una ricerca lineare per ogni parola che arriva per vedere se è già stata vista; il programma impiegherebbe un'eternità. (Più precisamente, il tempo di esecuzione crescerebbe quadraticamente col numero delle parole in input). Come possiamo organizzare i dati per far fronte in modo efficiente ad una lista arbitraria di parole?
Una soluzione è quella di tenere l'insieme delle parole incontrate fino ad un certo istante ordinate permanentemente, mettendo ogni parola al posto giusto nell'ordine in cui arriva. Questo non deve essere fatto facendo slittare le parole in un array lineare — anche questo impiega troppo tempo. Useremo invece una struttura di dati chiamata *albero binario*.
L'albero contiene un "nodo" per ogni parola; un nodo contiene

un puntatore al testo della parola
un contatore del numero delle occorrenze
un puntatore al nodo figlio di sinistra
un puntatore al nodo figlio di destra

Nessun nodo può avere più di due figli; può averne solo zero o uno.
I nodi sono disposti in modo che in ciascun nodo il sottoalbero sinistro contiene solo parole che sono minori della parola nel nodo, e il sottoalbero destro solo parole che sono maggiori. Per vedere se una nuova parola faccia già parte dell'albero, si parte dalla radice e si confronta la nuova parola con quella contenuta in quel nodo. Se sono uguali, alla domanda si risponde affermativamente. Se la parola nuova è minore della parola nell'albero la ricerca continua al figlio di sinistra, altrimenti si esamina il figlio di destra. Se non c'è un figlio nella direzione richiesta la nuova parola non è nell'albero, e di fatto il posto giusto per lei è proprio il figlio che manca. Questo processo di ricerca

è intrinsecamente ricorsivo, in quanto la ricerca che parte da qualsiasi nodo usa una ricerca da uno dei suoi figli. Analogamente, le procedure di ricerca e di stampa più naturali saranno ricorsive.
Tornando indietro alla definizione di nodo, si tratta chiaramente di una struttura con quattro componenti:

```
struct alb_nodo {        /* il nodo di base */
    char *parola;    /* punta al testo */
    int conta;  /* numero di occorrenze */
    struct alb_nodo *sinistra;    /* figlio di sinistra */
    struct alb_nodo *destra;      /* figlio di destra */
};
```

Questa dichiarazione "ricorsiva" di nodo può sembrare strana, ma è in realtà estremamente corretta. Non è permesso che una struttura contenga un'istanza di se stessa, ma

```
struct alb_nodo *sinistra;
```

dichiara che **sinistra** è un *puntatore* ad un nodo, non un nodo vero e proprio.
Il codice di tutto il programma è sorprendentemente contenuto, data la quantità di routines di supporto che abbiamo già scritto. Sono **getword**, per accettare una singola parola di input, e **alloc**, per sopperire allo spazio necessario per mettere da parte le parole.
La routine principale legge semplicemente le parole con **getword** e le inserisce nell'albero con la routine **albero**.

```
#define     MAXPAROLA     20

main()      /* conteggio della frequenza delle parole */
{
    struct alb_nodo *radice, *albero();
    char parola[MAXPAROLA];
    int t;

    radice = NULL;
    while ((t = getword(parola, MAXPAROLA)) != EOF)
        if (t == LETTERA)
            radice = albero(radice, parola);
    stampalbero(radice);
}
```

albero stessa viene di conseguenza. Una parola viene presentata da main al livello più alto dell'albero (la radice). Ad ogni stadio, quella parola viene confrontata con la parola già memorizzata nel nodo, ed è passata per competenza al sottoalbero destro o sinistro con una chiamata ricorsiva ad **albero**. Alla fine, la parola coincide con qualcosa già presente nell'albero (nel qual caso il contatore viene incrementato), o si incontra un puntatore nullo, che indica che deve essere creato ed aggiunto all'albero un nuovo

nodo. Se viene creato un nuovo nodo, albero ritorna un puntatore a quest'ultimo, che viene installato nel nodo padre.

```
struct alb_nodo *albero(p, w)     /* installa w in o sotto p */
struct alb_nodo *p;
char *w;
{
    struct alb_nodo *talloc();
    char *salvastr();
    int cond;

    if (p == NULL) {    /* e' arrivata una parola nuova */
        p = talloc();   /* alloca un nodo nuovo */
        p->parola = salvastr(w);
        p->conta = 1;
        p->sinistra = p->destra = NULL;
    } else if ((cond = strcmp(w, p->parola)) == 0)
        p->conta++;         /* parola ripetuta */
    else if (cond < 0)
    /* la parola minore va nel sottoalbero sinistro */
        p->sinistra = albero(p->sinistra, w);
    else
    /* quella maggiore nel sottoalbero destro */
        p->destra = albero(p->destra, w);
    return(p);
}
```

La memoria per il nuovo nodo è prelevata da una routine **talloc**, che è un adattamento della **alloc** scritta in precedenza. Ritorna un puntatore ad uno spazio libero adatto per ospitare il nodo di un albero. (Discuteremo ancora di questo tra poco). La nuova parola viene copiata da **salvastr** in una area riservata, il contatore viene inizializzato, e i due figli sono annullati. Questa parte di codice viene eseguita solo ai contorni dell'albero, quando viene aggiunto un nodo nuovo. Abbiamo omesso (incautamente per un programma commerciale) i controlli di errore sui valori restituiti da **salvastr** e **talloc**.
stampalbero per ogni nodo, stampa il sottoalbero sinistro (tutte le parole minori della parola corrente), quindi la parola corrente, quindi il sottoalbero destro (le parole maggiori). Se ci si sente dubbiosi sulla ricorsività, basta disegnare un albero e stamparlo poi con **stampalbero**; è una delle routines ricorsive più pulite che si possano trovare.

```
stampalbero(p)    /* stampa l'albero p ricorsivamente */
struct alb_nodo *p;
{
    if (p != NULL) {
        stampalbero(p->sinistra);
        printf("%4d %s\n", p->conta, p->parola);
        stampalbero(p->destra);
    }
}
```

Una nota per l'uso pratico: se l'albero diventa "sbilanciato" per il fatto che le parole non arrivano in ordine casuale, il tempo di esecuzione del programma può crescere troppo rapidamente. Come caso peggiore, se le parole sono già ordinate, questo programma farà una costosa simulazione di una ricerca sequenziale. Esistono generalizzazioni

dell'albero binario, gli alberi 2-3 e gli alberi **AVL**, che non soffrono di questo comportamento nel caso peggiore, ma non li descriveremo qui.
Prima di passare oltre questo esempio, vale anche la pena di fare una breve digressione su un problema legato agli allocatori di memoria. Chiaramente, è desiderabile che ci sia solo un allocatore di memoria in un programma, anche se per allocare oggetti di tipo differente. Ma se, ad esempio, un allocatore deve soddisfare a richieste di puntatori a caratteri e a strutture **alb_nodo**, sorgono due problemi. In primo luogo, come potrebbe andare incontro alla esigenza della maggior parte delle macchine che oggetti di un certo tipo devono soddisfare a restrizioni di allineamento (per esempio, gli interi devono spesso essere posti ad indirizzi dispari)? Secondariamente, quali dichiarazioni possono far fronte al fatto che **alloc** necessariamente debba restituire puntatori di tipo diverso?
Le condizioni di allineamento possono essere in generale risolte semplicemente, al costo di un certo spazio sprecato, solamente assicurando che l'allocatore restituisca sempre un puntatore che soddisfa a *tutte* le restrizioni. Per esempio, sul **PDP-11** è sufficiente che **alloc** ritorni un puntatore dispari, poiché qualsiasi tipo di oggetto può essere posto ad un indirizzo dispari. Il solo costo è un carattere sprecato per ogni richiesta di lunghezza dispari. Provvedimenti simili sono presi anche per altre macchine. In questo modo l'implementazione di **alloc** può non essere portabile, ma il suo uso lo è. La **alloc** del Capitolo 5 non garantisce alcun particolare allineamento; nel Capitolo 8 mostreremo come si può risolvere il problema correttamente.
Il problema della dichiarazione del tipo di **alloc** è impegnativo per qualsiasi linguaggio che prenda sul serio il controllo dei tipi. In C, la procedura migliore è quella di dichiarare che **alloc** ritorna un puntatore a **char**, e poi forzare il puntatore al tipo desiderato con un cast. Cioè, se **p** è dichiarato come

```
char *p;
```

allora

```
(struct alb_nodo *) p
```

lo converte in un puntatore ad **alb_nodo** nelle espressioni. In questo modo **talloc** è scritta come

```
struct alb_nodo *talloc()
{
    char *alloc();

    return((struct alb_nodo *) alloc(sizeof(struct alb_nodo)));
}
```

Questo è più del necessario per gli attuali compilatori, ma rappresenta la tendenza più sicura per il futuro.

Esercizio 6-4. Scrivere un programma che legga un programma C e stampi in ordine alfabetico i gruppi di nomi di variabili che sono identici nei primi 7 caratteri, ma differenti da quel punto in poi. (Assicurarsi che 7 sia un parametro).

Esercizio 6-5. Scrivere un cross-referencer essenziale: un programma che stampa l'elenco di tutte le parole di un documento e, per ciascuna parola, una lista dei numeri di linea in cui compare.

Esercizio 6-6. Scrivere un programma che stampi tutte le parole diverse del suo input ordinate per ordine decrescente di frequenza di occorrenza. Precedere ciascuna parola col suo contatore.

6.6 Ricerca tabellare

In questa sezione descriveremo le implicazioni di un package di ricerca tabellare per illustrare più aspetti delle strutture. Questo codice è tipico di ciò che può essere trovato nelle routine di gestione della symbol table di un macro processore o di un compilatore. Per esempio, considerate l'istruzione **#define** del C. Quando viene scandita una linea come

```
#define    SI    1
```

il nome **SI** e il testo di sostituzione 1 sono memorizzati in una tabella. In seguito, quando il nome **SI** compare in un'istruzione come

```
inparola = SI;
```

deve essere sostituito da 1.
Ci sono due routines principali che manipolano i nomi e il testo di sostituzione. **aggiungi (s,t)** registra il nome **s** ed il testo di sostituzione **t** in una tabella; **s** e **t** sono solamente stringhe di caratteri. **cerca (s)** cerca **s** nella tabella, e restituisce un puntatore al posto in cui la trova, o **NULL** se non c'era.
L'algoritmo usato è una ricerca hash — il nome viene convertito in un intero positivo piccolo, che verrà poi usato come indice di un array di puntatori. Il generico elemento dell'array punta all'inizio di una catena di blocchi che descrivono i nomi che hanno quel valore hash. È **NULL** se non ci sono nomi convertiti a quel valore hash.
Un blocco della catena è una struttura che contiene i puntatori al nome, al testo di sostituzione, e al blocco successivo nella catena. Un puntatore al blocco successivo che sia **NULL** determina la fine della catena.

```
struct plista { /* elemento base della tabella */
    char *nome;
    char *def;
    struct plista *prossimo; /* prossimo elemento della catena */
};
```

L'array di puntatori è semplicemente

```
#define    DIMHASH    100
static struct plista *tabella[DIMHASH]; /* tabella di puntatori */
```

La funzione hash, che è usata da **cerca** ed **aggiungi**, somma semplicemente i valori dei caratteri della stringa e ne calcola il resto della divisione per la dimensione della tabella. (Questo non è il migliore algoritmo possibile, ma ha dalla sua il merito della estrema semplicità).

```
hash(s) /* ottiene il valore hash della stringa s */
char *s;
{
    int hashval;

    for (hashval = 0; *s != '\0'; )
        hashval += *s++;
    return(hashval % DIMHASH);
}
```

Il processo di hashing produce un indice di partenza nell'array **tabella**; se la stringa deve essere trovata, sarà nella catena di blocchi che comincia là. La ricerca è effettuata da **cerca**. Se **cerca** trova che l'elemento è già presente, restituisce un puntatore che lo individua; in caso contrario, restituisce **NULL**.

```
struct plista *cerca(s) /* cerca s in tabella */
char *s;
{
    struct plista *np;

    for (np = tabella[hash(s)]; np != NULL; np = np->prossimo)
        if (strcmp(s, np->nome) == 0)
            return(np);     /* trovato */
    return(NULL);     /*non trovato */
}
```

aggiungi usa **cerca** per determinare se il nome da aggiungere sia già presente; se così fosse, la nuova definizione dovrebbe sovrapporsi alla vecchia.
Altrimenti, viene creato un elemento completamente nuovo.
aggiungi ritorna **NULL** se per qualsiasi ragione non c'è spazio sufficiente per inserire un nuovo elemento

```
struct plista *aggiungi(nome, def);  /* inserisce (nome, def) */
char *nome, *def;                    /* in tabella */
{
    struct plista *np, *cerca();
    char *salvastr(), *alloc();
    int hashval;

    if ((np = cerca(nome)) == NULL) { /* non trovata */
        np = (struct plista *) alloc(sizeof(*np));
        if (np == NULL)
            return(NULL);
        if ((np->nome = salvastr(nome)) == NULL)
            return(NULL);
        hashval = hash(np->nome);
        np->prossimo = tabella[hashval];
        tabella[hashval] = np;
    } else /* il nome c'e' gia' */
        free(np->def); /* rilascia la precedente definizione */
    if ((np->def = salvastr(def)) == NULL)
        return(NULL);
    return(np);
}
```

salvastr copia solamente la stringa data dal suo argomento in un posto sicuro ottenuto da una chiamata ad **alloc**. Ne abbiamo mostrato il codice nel Capitolo 5. Siccome chiamate ad **alloc** e **free** si possono presentare in ordine qualsiasi, e siccome l'allineamento conta, la semplice versione di **alloc** del Capitolo 5 non è adeguata al caso nostro; vedere i Capitoli 7 ed 8.

Esercizio 6-7. Scrivere una procedura che rimuova un nome e definizione dalla tabella gestita da **cerca** e **aggiungi**.

Esercizio 6-8. Implementare una semplice versione del processore di #**define** adatto per l'uso coi programmi C basato sulle routines di questa sezione. Si dovrebbe trovare utili anche **getch** e **ungetch**.

6.7 Campi

Quando lo spazio in memoria è prezioso, può essere necessario compattare diversi oggetti in una singola parola macchina; un uso molto comune è quello degli insiemi di flags di un bit in applicazioni come le symbol tables dei compilatori. Anche con dati di formato imposto esternamente, come le interface ai dispositivi hardware, è spesso richiesta la possibilità di trattare parti di parola.
Immaginate la parte di un compilatore che manipola una symbol table. Ciascun identificatore di un programma ha alcune informazioni ad esso associate, per esempio se è una parola chiave o meno, se è esterno e/o statico, e così via. Il modo più compatto di codificare tali informazioni è un insieme di flags di un bit in un unico **char** o **int**.
Il modo più comune per farlo è di definire un insieme di "maschere" corrispondenti alle posizioni di rilievo, come in

```
#define    KEYWORD    01
#define    EXTERNAL   02
#define    STATIC     04
```

(I numeri devono essere potenze del due). Allora l'accesso ai bits diventa una questione di "bit fiddling", con gli operatori di spostamento, mascheramento e complementazione descritti nel Capitolo 2.
Certi costrutti appaiono frequentemente:

```
flags|= EXTERNAL | STATIC;
```

attiva i bits di flags che indicano EXTERNAL e STATIC, mentre

```
flags &= ~(EXTERNAL | STATIC);
```

li disattiva, e

```
if ((flags & (EXTERNAL | STATIC)) == 0) ...
```

è vero se entrambi i bits sono a zero.
Sebbene queste locuzioni siano di rapida attuazione, come alternativa il C offre la

capacità di definire e accedere direttamente a campi nell'ambito di una parola senza l'uso degli operatori logici orientati al bit. Un *campo* è un insieme di bits adiacenti all'interno di un singolo **int**. La sintassi per la definizione e l'uso dei campi è basata sulle strutture. Per esempio, le linee di **#define** per la symbol table di cui sopra potrebbero essere sostituite dalla definizione di tre campi:

```
struct {
    unsigned is_keyword : 1;
    unsigned is_extern : 1;
    unsigned is_static : 1;
} flags;
```

Questo definisce una variabile chiamata **flags** che contiene tre campi di 1 bit. Il numero che segue i due punti rappresenta l'ampiezza del campo in bits. I campi sono dichiarati **unsigned** per enfatizzare il fatto che essi sono veramente quantità senza segno.
Ai singoli campi ci si riferisce come **flags.is_keyword, flags.is_extern**, ecc., proprio come agli altri membri di struttura. I campi si comportano come piccoli interi senza segno, e possono figurare nelle espressioni come gli altri interi. Gli esempi precedenti possono essere scritti in maniera più naturale come

```
flags.is_extern = flags.is_static = 1;
```

per attivare i bits;

```
flags.is_extern = flags.is_static = 0;
```

per disattivarli, e

```
if (flags.is_extern == 0 && flags.is_static == 0) ...
```

per controllarli.
Un campo non può eccedere il limite di un intero; se la sua ampiezza lo causasse, il campo verrebbe allineato al successivo indirizzo adatto per un intero. Non è necessario che ai campi venga dato un nome; i campi senza nome (solamente i due punti e un numero) vengono usati come riempitivi. La lunghezza speciale 0 serve per forzare l'allineamento al successivo indirizzo di intero.
Ci sono un certo numero di particolarità riguardo ai campi. Forse la più significativa è che i campi sono assegnati da sinistra a destra su alcune macchine e da destra a sinistra su altre, come riflesso della natura di hardware diversi. Questo significa che sebbene i campi siano piuttosto utili per mantenere dati definiti internamente, il problema di quale estremo venga prima deve essere accuratamente considerato quando si isolano dati definiti esternamente.
Altre restrizioni da tenere a mente: i campi sono privi di segno; possono essere immagazzinati solo in **int** (o, ugualmente in **unsigned**); non sono arrays; non hanno indirizzo, quindi l'operatore **&** non può essere loro applicato.

6.8 Unioni

Una *unione* è una variabile che può contenere, in istanti differenti, oggetti di tipo e dimensioni diverse; il compilatore tiene traccia dei requisiti di dimensione e allineamento. Le unioni danno modo di manipolare tipi di dati diversi in una sola area di memoria, senza inserire alcuna informazione dipendente dalla macchina nel programma.
Come esempio, ancora da una symbol table di un compilatore, supponiamo che delle costanti possano rappresentare puntatori a numeri interi, in virgola mobile oppure a caratteri. Il valore di una certa costante deve essere memorizzato in una variabile del tipo appropriato, anche se sarebbe più vantaggioso per la gestione della tabella se il valore occupasse la stessa quantità di memoria e fosse memorizzato nello stesso posto indipendentemente dal suo tipo. Questo è lo scopo dell'unione — mettere a disposizione una sola variabile che possa correttamente contenere uno qualsiasi dei diversi tipi. Come per i campi, la sintassi è basata sulle strutture.

```
union u_tag {
    int ival;
    float fval;
    char *pval;
} uval;
```

La variabile **uval** sarà abbastanza capace da contenere il tipo di occupazione maggiore dei tre, indipendentemente dalla macchina su cui è compilato — il codice è indipendente dalle caratteristiche dell'hardware. Ciascuno di questi tipi può essere assegnato ad **uval** e quindi usato in espressioni, sempre che l'uso sia coerente; il tipo da ottenere deve essere quello assegnato più recentemente. È responsabilità del programmatore tenere traccia di quale tipo sia contenuto coorentemente in un'unione; i risultati dipendono dalla macchina se un oggetto viene memorizzato come di un certo tipo ed estratto come di un altro.
Sintatticamente, ai membri di una struttura si accede come

nome-di-unione.membro

oppure

puntatore-ad-unione –> membro

proprio come nelle strutture. Se viene usata una variabile **utipo** per tenere traccia del tipo correntemente contenuto in **uval**, si può vedere un codice di questo tipo

```
if (utipo == INT)
    printf("%d\n", uval.ival);
else if (utipo == FLOAT)
    printf("%f\n", uval.fval);
else if (utipo == STRINGA)
    printf("%s\n", uval.pval);
else
    printf("tipo errato %d in utipo\n", utipo);
```

Le unioni possono intervenire nell'ambito di strutture e di arrays e viceversa. La notazione per accedere ad un membro di unione che fa parte di una struttura (o viceversa) è identica a quella usata per le strutture nidificate. Per esempio, l'array di strutture definito da

```
struct {
    char *nome;
    int flags;
    int utipo;
    union {
        int ival;
        float fval;
        char *pval;
    } uval;
} symtab[NSYM];
```

Alla variabile **ival** ci si può riferire come

```
symtab[i].uval.ival
```

ed al primo carattere della stringa **pval** come

```
*symtab[i].uval.pval
```

In effetti, un'unione è una struttura in cui tutti i membri hanno distanza zero, la struttura è grande abbastanza per contenere il membro più "largo" e l'allineamento è appropriato per tutti i tipi dell'unione. Come per le strutture, le sole operazioni a tutt'oggi permesse sulle unioni sono accederne ad un membro e ricavarne l'indirizzo; sulle unioni non si possono operare assegnamenti, non si può passarle a funzioni, e neppure possono essere restituite da funzioni. I puntatori alle unioni possono essere usati in maniera identica ai puntatori di struttura.
L'allocatore di memoria del capitolo 8 mostra come un'unione possa essere usata per forzare l'allineamento di una variabile ad un tipo particolare di soglia di memoria.

6.9 Typedef

Il C dispone di una caratteristica chiamata **typedef** per creare nuovi nomi di tipi di dati. Per esempio, la dichiarazione

```
typedef int AMPIEZZA;
```

fa del nome **AMPIEZZA** un sinonimo di **int**. Il "tipo" **AMPIEZZA** può essere usato in dichiarazioni, casts, ecc., esattamente con le stesse modalità con cui può essere usato il tipo **int**:

```
AMPIEZZA    amp, maxamp;
AMPIEZZA    *ampiezze[];
```

Analogamente, la dichiarazione

```
typedef char *STRINGA;
```

fa di **STRINGA** un sinonimo di **char** * o puntatore a carattere, che può essere usato successivamente in dichiarazioni come

```
STRINGA p, lineptr[LINEE], alloc();
```

È da notare che il tipo che si dichiara in **typedef** appare nella posizione di un nome di variabile, non subito dopo la parola **typedef**. Sintatticamente, **typedef** è come le classi di memorizzazione **extern**, **static**, ecc. Abbiamo anche usato le maiuscole per enfatizzare i nomi.
Come esempio più complesso, potremmo usare **typedef** per i tre nodi incontrati precedentemente in questo capitolo:

```
typedef struct alb_nodo {     /* il nodo di base */
    char *parola;     /* punta al testo */
    int conta;        /* numero di ricorrenze */
    struct alb_nodo *sinistra;     /* figlio di sinistra */
    struct alb_nodo * destra;     /* figlio di destra */
} NODOALBERO, *ALBEROPTR;
```

Ciò crea due nuove parole riservate di tipo chiamate **NODOALBERO** (una struttura) e **ALBEROPTR** (un puntatore a struttura). Di conseguenza la routine **talloc** potrebbe diventare

```
ALBEROPTR talloc()
{
    char *alloc();

    return((ALBEROPTR) alloc(sizeof(NODOALBERO)));
}
```

Bisogna enfatizzare il fatto che una dichiarazione **typedef** non crea in nessun senso un nuovo tipo; aggiunge solamente un nuovo nome per qualche tipo già esistente. Non c'è nemmeno una nuova semantica: le variabili dichiarate in questo modo hanno esattamente le stesse proprietà delle variabili le cui dichiarazioni sono enunciate esplicitamente. In effetti, **typedef** è simile a **#define**, ad eccezione del fatto che essendo interpretata dal compilatore, può far fronte a sostituzioni di testo che vanno oltre le capacità del macro processore del C. Per esempio,

```
typedef int (*PFI)();
```

crea il nuovo tipo **PFI**, che sta per "puntatore ad una funzione che ritorna un **int**," che può essere usato in contesti come

```
PFI strcmp, numcmp, swap;
```

del programma di ordinamento del Capitolo 5.

Ci sono due ragioni principali per usare le dichiarazioni **typedef**. La prima è per parametrizzare un programma nei confronti di problemi di portabilità. Se si usano **typedef** per tipi di dati che potrebbero essere dipendenti dalla macchina, deve essere cambiata solo la **typedef** quando il programma è trasportato. Una situazione comune è quella di usare nomi di **typedef** per varie quantità intere, e quindi creare una appropriata scelta di **short**, **int** e **long** per diverse macchine ospiti.
Il secondo scopo di **typedef** è quello di mettere a disposizione una migliore documentazione dei programmi — un tipo che si chiama **ALBEROPTR** può essere più facile da capire di un altro dichiarato solo come un puntatore ad una struttura complicata.
Infine, c'è sempre la possibilità che in futuro il compilatore o qualche altro programma come *lint* possano fare uso delle informazioni contenute in **typedef** per effettuare dei controlli in più sul programma.

CAPITOLO 7

INPUT ED OUTPUT

Le strutture predisposte all'input ed output non fanno parte del linguaggio C, per questo motivo non sono ancora state trattate a fondo nella nostra esposizione. Comunque, i programmi veri e propri interagiscono col loro ambiente in modi assai più complessi di quelli descritti precedentemente. In questo capitolo descriveremo "la libreria standard di **I/O**", un insieme di funzioni progettate per provvedere ad un sistema standard di **I/O** per i programmi C. Si richiede che le funzioni presentino una opportuna interfaccia per la programmazione, riflettendo tuttavia solo le operazioni che possono essere fornite da quasi tutti i moderni sistemi operativi. Le routine sono abbastanza efficienti e raramente gli utenti avranno l'esigenza di aggirarle per motivi "di efficienza", indipendentemente dalla portata dell'applicazione. Infine, si intende che le routine siano "portatili", nel senso che esisteranno in una forma compatibile su ogni sistema in cui esiste il C, ed i programmi che limitano le interazioni col sistema alle facilitazioni fornite dalla libreria standard possano essere spostati da un sistema all'altro senza cambiamenti essenziali. Non tenteremo qui di descrivere l'intera libreria di **I/O**; siamo più interessati a mostrare gli elementi essenziali per scrivere programmi C che interagiscono con le condizioni ambientali del proprio sistema operativo.

7.1 Accesso alla Libreria Standard

Ogni file sorgente che fa riferimento a una funzione della libreria standard deve contenere la linea

```
#include <stdio.h>
```

vicino all'inizio. Il file **stdio.h** definisce alcune macro e variabili usate dalla libreria di **I/O**. L'uso delle parentesi angolari < e > al posto degli usuali doppi apici dirige il compilatore a cercare il file in una directory contenente informazioni iniziali standard (in **UNIX**, tipicamente */usr/include*).
Inoltre, può essere necessario specificare la libreria esplicitamente quando si carica il programma; per esempio, sul sistema **UNIX** del **PDP/11**, il comando per compilare il programma sarebbe

cc *files sorgenti, ecc.* –lS

in cui **−lS** indica di caricare dalla libreria standard. (Il carattere **l** è la lettera elle).

7.2 Standard Input ed Output — Getchar e Putchar

Il meccanismo più semplice di input è leggere un carattere per volta dallo "standard input", generalmente il terminale dell'utente, con **getchar**. **getchar()** ritorna il successivo carattere di input ogni volta che viene chiamata. Nella maggior parte degli ambienti che supportano il C, può essere sostituito un file al posto del terminale usando la convenzione <: se un programma *prog* usa **getchar**, allora la linea di comando

```
prog <infile
```

fa in modo che *prog* legga **infile** al posto del terminale. Lo scambio dell'input viene effettuato in una maniera tale per cui *prog* stesso non è consapevole del cambiamento; in particolare, la stringa "<**infile**" non viene inclusa negli argomenti della linea di comando in **argv**. Il cambiamento dell'input è invisibile anche se l'input arriva da un altro programma attraverso il meccanismo della pipe; la linea di comando

```
altroprog | prog
```

esegue i due programmi *altroprog* e *prog* e fa sì che lo standard input per *prog* provenga dallo standard output di *altroprog*.
getchar ritorna il valore **EOF** quando incontra la fine del file su qualsiasi input stia leggendo. La libreria standard definisce la costante simbolica **EOF** uguale a **−1** (con un **#define** nel file **stdio.h**), ma le verifiche dovrebbero essere scritte in termini di **EOF**, non **−1**, in modo da essere indipendente dal valore specifico.
Per l'output, **putchar(c)** manda il carattere **c** nello "standard output", che anch'esso è per il default il terminale. L'output può essere diretto su di un file usando >: se *prog* usa **putchar**,

```
prog > outfile
```

scriverà lo standard output su **outfile** al posto del terminale. Sul sistema **UNIX** può anche essere usata una pipe:

```
prog | unaltroprog
```

manda lo standard output di *prog* nello standard input di *unaltroprog*. Ancora una volta *prog* non si rende conto della redirezione.
Anche l'output prodotto da **printf** trova la propria via di standard output, potendo così alternare chiamare a **putchar** e a **printf**.
Un sorprendente numero di programmi legge un solo flusso di input e scrive un solo flusso di output; per tali programmi, l'**I/O** con **getchar**, **putchar** e **printf** sono estremamente adatti e certamente sufficienti per iniziare.
Questo vale soprattutto per quando esiste la possibilità di redirezione su file e la facilitazione della pipe per connettere l'output di un programma con l'input del successi-

vo. Per esempio, si consideri il programma *lower*, che trasforma il proprio input in lettere minuscole:

```
#include <stdio.h>

main()    /* converte l'input in lettere minuscole */
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(isupper(c) ? tolower(c) : c);
}
```

In realtà le "funzioni" **isupper** e **tolower** sono delle macro definite in **stdio.h**. La macro **isupper** verifica se il suo argomento è una lettera maiuscola ritornando non-zero se lo è e zero se non lo è. La macro **tolower** converte una lettera maiuscola in minuscola. Indipendentemente da come queste funzioni vengono implementate su una particolare macchina, il loro comportamento esterno è sempre lo stesso, così i programmi che le usano non hanno bisogno di conoscere il set di caratteri usato.
Per convertire più files si può usare un programma come la utility di **UNIX** *cat* per riunire i files:

```
cat file1 file2 ... | lower >output
```

e in questo modo si evita di sapere come accedere ai files da un programma (*cat* viene presentato più tardi in questo capitolo).
Per inciso, nella libreria standard di **I/O** le "funzioni" **getchar** e **putchar** possono veramente essere macro, ed evitare così l'aggravio di una chiamata di funzione per ogni carattere. Vedremo come questo viene ottenuto nel Capitolo 8.

7.3 Output Formattato — Printf

Le due routines **printf** per l'output e **scanf** per l'input (prossimo paragrafo) permettono la conversione verso e dalle rappresentazioni in carattere di quantità numeriche. Permettono inoltre la generazione o l'interpretazione di linee formattate. Abbiamo usato **printf** informalmente in tutti i precedenti capitoli; ne diamo ora una descrizione più completa e precisa.

```
printf(control, arg1, arg2, ...)
```

printf converte, formatta e stampa i suoi argomenti sullo standard output sotto il controllo della stringa **control**. La stringa di controllo contiene due tipi di oggetti: caratteri comuni che vengono semplicemente copiati nel flusso di output e specifiche di conversione, ognuna delle quali determina la conversione e la stampa del successivo argomento di **printf**.

Ogni specifica di conversione viene introdotta dal carattere **%** e viene terminata da un carattere di conversione. Tra **%** ed il carattere di conversione può esserci:

Un segno meno, che specifica l'allineamento a sinistra nel suo campo dell'argomento convertito.

Una stringa di cifre che specificano la lunghezza minima del campo. Il numero convertito verrà stampato in un campo che sarà almeno di questa larghezza, e più largo se necessario. Se l'argomento convertito ha meno caratteri della larghezza del campo, verrà allungato a sinistra (o a destra, se è stato dato l'indicatore di allineamento a sinistra) per ottenere la larghezza del campo. Normalmente il carattere riempitivo è blank, è zero se l'ampiezza del campo è stata specificata con uno zero in testa (questo zero non implica una larghezza di campo ottale).

Un punto, che separa la larghezza del campo dalla successiva stringa di cifre.

Una stringa di cifre (la precisione) che specifica il massimo numero di caratteri che devono essere stampati da una stringa, o il numero di cifre da stampare alla destra dal punto decimale di un **float** o **double**.

Un modificatore di larghezza **l** (lettera elle) che indica che il corrispondente dato è un **long** invece che un **int**.

I caratteri di conversione ed il loro significato sono:

d L'argomento viene convertito nella notazione decimale.
o L'argomento viene convertito in una notazione ottale senza segno (senza lo zero in testa).
x L'argomento viene convertito in una notazione esadecimale senza segno (senza **0x** in testa).
u L'argomento viene convertito in una notazione decimale senza segno.
c Si considera che l'argomento sia un singolo carattere.
s L'argomento è una stringa; i caratteri della stringa vengono stampati finché non viene incontrato un carattere nullo o finché non viene raggiunto il numero dei caratteri indicati nella specifica di precisione.
e Si considera che l'argomento sia **float** o **double** e viene convertito nella notazione decimale con la forma **[−]m.nnnnnnE[+−]xx** in cui la lunghezza della stringa degli **n** viene specificata dalla precisione. La precisione di default è 6.
f Si considera che l'argomento sia **float** o **double** e viene convertito nella notazione decimale nella forma **[−]mmm.nnnnn** in cui la lunghezza della stringa di **n** viene specificata dalla precisione. La precisione di default è 6. Si noti che la precisione non determina il numero di cifre significative stampate nel formato **f**.
g Usa **%e** o **%f**, quello che è più corto; gli zeri non significativi non vengono stampati.

Se il carattere dopo **%** non è un carattere di conversione, questo carattere viene stampato; così **%** può essere stampato con **%%**.
La maggior parte dei formati di conversione sono ovvi, e sono descritti nei precedenti capitoli. La precisione collegata alle stringhe è un'eccezione. La tavola seguente mostra

gli effetti di una varietà di specificazioni nello stampare "salve, mondo" (12 caratteri). Abbiamo inserito i due punti intorno ad ogni campo per mostrarne l'estensione.

```
:%10s:          :salve, mondo:
:%-10s:         :salve, mondo:
:%20s:          :        salve, mondo:
:%-20s:         :salve, mondo        :
:%20.10s:       :          salve, mon:
:%-20.10s:      :salve, mon          :
:%.10s:         :salve, mon:
```

Un avvertimento: **printf** usa il suo primo argomento per decidere quanti argomenti seguono e di che tipo sono. Si avranno risposte assurde se non ci saranno argomenti sufficienti o se questi saranno del tipo sbagliato.

Esercizio 7-1. Scrivere un programma che stampi un input arbitrario in modo ragionevole. Come minimo, dovrebbe stampare caratteri non visualizzabili in ottale o esadecimale (in accordo con la consuetudine locale) e spezzare linee lunghe.

7.4 Input Formattato — Scanf

La funzione **scanf** è l'analoga per l'input di **printf**, fornendo molte facilitazioni di conversione dello stesso tipo ma nella direzione opposta.

```
scanf(control, arg1, arg2, ...)
```

scanf legge i caratteri da standard input, li interpreta secondo il formato specificato in **control** e immagazzina i risultati negli argomenti rimanenti. L'argomento control è descritto sotto; gli altri argomenti, *ognuno dei quali dev'essere un puntatore*, indicano dove deve essere memorizzato il corrispondente input convertito.
La stringa di controllo contiene generalmente le specifiche di conversione, usate per guidare l'interpretazione delle sequenze di input. La stringa di controllo può contenere:

Blank, tab o newline ("caratteri di spazio bianchi"), che vengono ignorati.

Caratteri ordinari (non **%**) che dovrebbero coincidere con il prossimo carattere diverso da un carattere bianco nel flusso di input.

Specifiche di conversione, consistenti del carattere **%**, un carattere opzionale * che sopprime l'assegnamento, un numero opzionale che specifica la massima ampiezza del campo ed un carattere di conversione.

Una specifica di conversione guida la conversione del successivo campo di input. Normalmente il risultato viene posto nella variabile puntata dal corrispondente argomento. Comunque, se tramite il carattere * viene soppresso l'assegnamento, il campo di input viene semplicemente saltato; non viene fatto alcun assegnamento. Un campo di input viene definito come una stringa di caratteri diversi da spazi bianchi; si estende fino al successivo carattere di spazio bianco o fino a che è stata raggiunta, se specificata, l'ampiezza del campo. Questo implica che **scanf** legge oltre i limiti della linea per cercare il proprio input poiché i newline sono spazi bianchi.

Il carattere di conversione indica l'interpretazione del campo di input; l'argomento corrispondente dev'essere un puntatore, come richiesto dalla semantica della chiamata per valore del C. Sono validi i seguenti caratteri di conversione:

- d Nell'input si aspetta un intero decimale; l'argomento corrispondente dev'essere un puntatore ad un intero.
- o Nell'input si aspetta un intero ottale (con o senza lo zero in testa); l'argomento corrispondente dev'essere un puntatore ad un intero.
- x Nell'input si aspetta un intero esadecimale (con o senza lo **0x** in testa); l'argomento corrispondente dev'essere un puntatore ad un intero.
- h Nell'input si aspetta un intero **short**; l'argomento corrispondente dev'essere un puntatore ad un intero **short**.
- c Si aspetta un singolo carattere; l'argomento corrispondente dev'essere un puntatore a carattere; il successivo carattere di input viene posto nel luogo indicato. In questo caso viene soppresso il normale salto dei caratteri di spazi bianchi; per leggere il successivo carattere non bianco, si deve usare **%1s**.
- s Si aspetta una stringa di caratteri; l'argomento corrispondente dev'essere un puntatore a carattere che punta ad un array di caratteri abbastanza ampio da accettare la stringa e un **\0** di termine che verrà aggiunto.
- f Si aspetta un numero in floating point; l'argomento corrispondente dev'essere un puntatore a **float**. Il carattere di conversione **e** è un sinonimo di **f**. Il formato di input per i **float** è un segno opzionale, una stringa di numeri che può contenere un punto decimale ed un opzionale campo esponenziale contenente una **E** o **e** seguita da un intero possibilmente con segno.

I caratteri di conversione **d, o** e **x** possono essere preceduti da **l** (lettera elle) per segnalare che nell'elenco di argomenti appare un puntatore a **long** invece che **int**. Allo stesso modo, i caratteri di conversione **e** o **f** possono essere preceduti da **l** per indicare che nell'elenco di argomenti appare un puntatore a **double** invece che a **float**.
Per esempio, la chiamata

```
int i;
float x;
char nome[50];
scanf("%d %f %s", &i, &x, nome);
```

con la linea di input

```
25   54.32E-1  Thompson
```

assegnerà il valore **25** ad **i**, il valore di **5,432** ad **x**, e la stringa **"Thompson"**, terminata da **\0**, a **nome**. I tre campi di input possono essere separati da quanti blank, tab o newline si desidera. La chiamata

```
int i;
float x;
char nome[50];
scanf("%2d %f %*d %2s", &i, &x, nome);
```

con l'input

```
56789 0123 45a72
```

assegnerà **56** ad **i**, **789.0** ad **x**, salta oltre **0123** e colloca la stringa **"45"** in **nome**. La chiamata successiva a qualsiasi routine di input inizierà la ricerca dalla lettera a. In questi due esempi **nome** è un puntatore perciò *non* dev'essere preceduto da **&**.
Può fare un altro esempio, il rudimentale calcolatore del Capitolo 4 può ora essere scritto con **scanf** per ottenere una conversione dell'input:

```
#include <stdio.h>

main()   /* rudimentale calcolatore da tavolo */
{
    double somma, v;

    somma = 0;
    while (scanf("%lf", &v) != EOF)
        printf("\t%.2f\n", somma += v);
}
```

scanf si ferma quando ha esaurito le stringhe di controllo o quando qualche input fallisce l'accoppiamento con la specificazione di controllo. Come proprio valore ritorna il numero di oggetti di input che sono stati regolarmente accoppiati e assegnati. Questo può essere usato per sapere quanti oggetti di input sono stati trovati. Alla fine del file, viene ritornato **EOF**; è da notare che ciò è diverso da 0, che significa che il successivo carattere di input non si accorda con la prima specificazione nella stringa di controllo. Una successiva chiamata a **scanf** riprende la ricerca immediatamente dopo l'ultimo carattere già ritornato.
Un ultimo avvertimento: gli argomenti di **scanf** *devono* essere puntatori. L'errore di gran lunga più comune è quello di scrivere

```
scanf("%d", n);
```

al posto di

```
scanf("%d", &n);
```

7.5 Conversione di Formato in Memoria

Le funzioni **scanf** e **printf** hanno delle sorelle chiamate **sscanf** e **sprintf** che eseguono le stesse conversioni ma operano su stringhe invece che su file. Il formato generale è

```
sprintf(string, control, arg1, arg2, ...)
sscanf(string, control, arg1, arg2, ...)
```

sprintf formatta gli argomenti in **arg1**, **arg2**, ecc., sempre secondo **control**, ma pone il risultato in **string** invece che sullo standard output. Naturalmente **string** dovrebbe

essere lunga abbastanza da ricevere il risultato. Per esempio se **nome** è un array di caratteri ed **n** è un intero, allora

```
sprintf(nome, "temp%d", n);
```

crea una stringa della forma **temp***nnn* in **nome**, in cui *nnn* è il valore di **n**.
sscanf effettua la conversione inversa — scorre la stringa secondo il formato in **control**, e assegna i valori risultanti ad **arg1**, **arg2**, ecc. Questi argomenti devono essere puntatori. La chiamata

```
sscanf(nome, "temp%d", &n);
```

definisce **n** come il valore della stringa di cifre che segue **temp** all'interno di **nome**.

Esericzio 7-2. Riscrivere il calcolatore da tavolo del Capitolo 4 usando **scanf** e/o **sscanf** per ottenere la conversione dell'input e dei numeri.

7.6 Accesso a File

Tutti i programmi scritti finora hanno letto lo standard input e scritto sullo standard output, che abbiamo stabilito essere come per magia predefiniti per un programma dal sistema operativo locale.
Il prossimo passo nell'**I/O** è di scrivere un programma che accede ad un file che *non* sia già connesso al programma. Un programma che illustra chiaramente la necessità di tali operazioni è *cat*, che concatena nello standard output un insieme di files chiamati per nome. **cat** viene usato per stampare files sul terminale e per uso generale come raccoglitore di input per i programmi che non hanno la capacità di accedere ai files per nome. Per esempio, il comando

```
cat x.c y.c
```

stampa il contenuto dei files **x.c** e **y.c** nello standard output.
Il problema è come fare in modo che siano letti i files chiamati per nome — cioè, come collegare i nomi esterni forniti dall'utente con le istruzioni che effettivamente leggeranno i dati.
Le regole sono semplici. Un file, prima di poter essere letto o scritto, dev'essere **aperto** con la funzione della libreria standard **fopen**. **fopen** prende un nome esterno (come **x.c** o **y.c**, effettua alcuni controlli e scambi con il sistema operativo (dettagli che non ci riguardano), e ritorna un nome interno che dev'essere usato nelle successive letture o scritture del file.
In realtà il nome interno è un puntatore chiamato *puntatore al file*, ad una struttura che contiene informazioni circa il file, come la locazione di un buffer, la posizione corrente del carattere nel buffer, se si sta leggendo o scrivendo il file e cose simili. Gli utenti non hanno bisogno di conoscere i dettagli, poiché parte delle definizioni standard per l'**I/O** ottenute da **stdio.h**, sono la definizione di una struttura chiamata **FILE**. L'unica dichiarazione necessaria per un puntatore a file è del tipo:

```
FILE *fopen(), *fp;
```

Ciò afferma che **fp** è un puntatore ad un **FILE** e **fopen** ritorna un puntatore ad un **FILE**. Da notare che **FILE** è il nome di un tipo, come **int**, non un identificatore di struttura; è implementato come **typedef**. (I dettagli su come funziona tutto ciò nel sistema **UNIX** vengono forniti nel Capitolo 8).
La chiamata vera e propria a **fopen** in un programma é

```
fp = fopen(nome, modo);
```

Il primo argomento di **fopen** è il nome del file, come stringa di caratteri. Il secondo argomento è la *modalità*, anch'essa come stringa di caratteri, che indica come si intende usare il file. Modalità disponibili sono lettura (**"r"**, sta per read), scrittura (**"w"**, sta per write) o aggiunta (**"a"**, sta per append).
Se si apre un file che non esiste per la scrittura o l'aggiunta, esso viene creato (se possibile). Se si apre un file già esistente in scrittura, il vecchio contenuto viene cancellato. È un errore cercare di leggere un file che non esiste, così come ci possono essere anche altre cause di errore. (Come tentare di leggere un file a cui non si ha accesso). Se c'è un errore, **fopen** ritorna il valore di puntatore nullo **NULL** (che per comodità è anch'esso definito in **stdio.h**).
Una volta che il file è stato aperto, si ha bisogno di un modo per leggerlo o scriverlo. Ci sono varie possibilità, delle quali **getc** e **putc** sono le più semplici. **getc** ritorna il carattere successivo da un file; necessita del puntatore al file per far sapere quale file sia. Così

```
c = getc(fp)
```

manda in **c** il carattere successivo dal file indicato con **fp**, ed **EOF** quando incontra fine file.
putc è l'inverso di **getc**:

```
putc(c, fp)
```

manda il carattere **c** nel file **fp** e ritorna **c**. Come **getchar** e **putchar**, **getc** e **putc** possono essere macro invece che funzioni.
Quando un programma inizia la sua esecuzione, vengono aperti automaticamente tre files e vengono forniti i puntatori ai file per essi. Questi files sono lo standard input, lo standard output e lo standard error output; i corrispondenti puntatori ai file sono chiamati **stdin**, **stdout** e **stderr**. Normalmente sono tutti connessi al terminale, ma **stdin** e **stdout** possono essere rediretti su files o pipe come descritto nel paragrafo 7.2.
getchar e **putchar** possono essere definiti in termini di **getc putc** e **stdout** come segue:

```
#define getchar()   getc(stdin)
#define putchar(c)  putc(c, stdout)
```

Per input o output formattato di files, si possono usare le funzioni **fscanf** e **fprintf**. Esse sono identiche a **scanf** e **printf**, salvo che il primo argomento è un puntatore a file che specifica il file che dev'essere letto o scritto; la stringa di controllo è il secondo argomento.
Chiariti questi preliminari, siamo ora in grado di scrivere il programma cat per concatenare files. Il modello base è uno che è stato trovato conveniente per molti programmi: se

ci sono argomenti sulla linea di comando, vengono attivati in ordine. Se non ci sono argomenti, viene attivato lo standard input.
In questo modo il programma può essere usato da solo o come parte di un processo più ampio.

```
#include <stdio.h>

main(argc, argv)  /* cat: concatena files */
int argc;
char *argv[];
{
    FILE *fp, *fopen();

    if (argc == 1) /* nessun argomento; copia lo standard input */
        filecopy(stdin);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                printf("cat: non posso aprire %s\n", *argv);
                break;
            } else {
                filecopy(fp);
                fclose(fp);
            }
}

filecopy(fp)   /* copia il file fp nello standard output */
FILE *fp;
{
    int c;

    while ((c = getc(fp)) != EOF)
        putc(c, stdout);
}
```

I puntatori a file **stdin** e **stdout** sono predefiniti nella libreria di **I/O** come lo standard input e lo standard output; possono essere usati ovunque ci sia un oggetto di tipo **FILE** *. Comunque essi sono costanti, non variabili per cui non bisogna tentare di assegnare ad essi un qualsiasi valore.
La funzione **fclose** è l'inverso di **fopen**; rompe la connessione tra il puntatore al file ed il nome esterno che è stato stabilito da **fopen**, rilasciando il puntatore al file disponibile per un altro file. Poiché la maggior parte dei sistemi operativi hanno alcuni limiti sul numero di files aperti simultaneamente che può avere un programma, è un buon metodo rilasciare gli oggetti quando non se ne ha più bisogno, come abbiamo fatto in *cat*. **fclose** ha anche un altro scopo in un file di output — rilascia il buffer in cui **putc** sta accumulando l'output. (Quando un programma termina normalmente, **fclose** viene chiamata automaticamente per ogni file aperto).

7.7 Gestione degli Errori — Stderr e Exit

La gestione degli errori in *cat* non è ideale. Il guaio è che se non si può accedere ad uno dei files per qualsiasi motivo, la diagnostica viene stampata alla fine dell'output concatenato. Ciò è accettabile se quell'output sta andando su un terminale, ma non lo è se sta andando in un file o in un altro programma attraverso una pipeline.
Per gestire meglio questa situazione, viene assegnato a ciascun programma un secondo file di output, chiamato **stderr**, nello stesso modo in cui vengono assegnati **stdin** e **stdout**. Se possibile, l'output scritto su **stderr** compare sul terminale dell'utente anche se è stato rediretto lo standard output.
Rivediamo *cat* scrivendo i suoi messaggi di errore nel file di standard error.

```
#include <stdio.h>

main(argc,argv)   /* cat: concatena files */
int argc;
char *argv[];
{
    FILE *fp, *fopen();

    if (argc == 1)  /*nessun argomento; copia lo standard input */
        filecopy(stdin);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL)  {
                fprintf(stderr,
                    "cat: non posso aprire %s\n", *argv);
                exit(1);
            } else {
                filecopy(fp);
                fclose(fp);
            }
    exit(0);
}
```

Il programma segnala gli errori in due modi. L'output di diagnositica prodotto dal **fprintf** va nello **stderr**, così trova il suo sbocco nel terminale dell'utente invece di sparire in una pipeline o in un file di output.
Il programma usa anche la funzione della libreria standard **exit**, che, quando viene chiamata, termina l'esecuzione del programma. L'argomento di **exit** è disponibile per qualsiasi processo che abbia chiamato quest'ultimo, in modo che il successo o il fallimento del programma possa essere verificato da un altro programma che usi il primo come sottoprocesso. Per convenzione, un valore di ritorno uguale a 0 segnala che va tutto bene, ed i vari valori diversi da zero segnalano una situazione anomala.
exit chiama **fclose** per ogni file di output aperto per rilasciare ogni output bufferizzato, poi chiama una routine detta **_exit**. La funzione **_exit** causa immediatamente la fine senza alcun rilascio del buffer; naturalmente se si vuole si può chiamare direttamente.

7.8 Input e Output di Linea

La libreria standard fornisce una routine chiamata **fgets** che è molto simile alla funzione **getline** che abbiamo usato in tutto il libro. La chiamata

```
fgets(line, MAXLINEA, fp)
```

legge la successione linea di input (compreso il newline) dal file **fp** all'interno dell'array di caratteri **line**; verranno letti al massimo **MAXLINEA-1** caratteri. La linea risultante verrà terminata con **\0**. Normalmente **fgets** ritorna **line**; alla fine del file ritorna **NULL**. (Il nostro **getline** ritorna la lunghezza della linea e zero per fine file).
Per l'output, la funzione **fputs** scrive una stringa (che non deve necessariamente contenere il newline) in un file:

```
fputs(line, fp)
```

Per dimostrare che non c'è niente di magico circa funzioni come **fgets** e **fputs**, le riportiamo qui, copiate direttamente dalla libreria standard di I/O:

```
#include <stdio.h>

char *fgets(s, n, iop)  /* prende al massimo n caratteri da iop */
char *s;
int n;
register FILE *iop;
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs = '\0';
    return((c == EOF && cs == s) ? NULL : s);
}
```

```
fputs(s, iop)  /* manda la stringa s nel file iop */
register char *s;
register FILE *iop;
{
    register int c;

    while (c = *s++)
        putc(c, iop);
}
```

Esercizio 7-3. Scrivere un programma per confrontare due files, stampando la prima linea e carettere in cui questi differiscono.

Esercizio 7-4. Si modifichi il programma di ricerca pattern del Capitolo 5 in maniera tale che prenda il proprio input da un insieme di files con nome o, se non compaiono nomi di files come argomenti, dallo standard input. Si potrebbe stampare stampare il nome del file quando si trova una linea che incontra il pattern?

Esercizio 7-5. Scrivere un programma che stampi un insieme di files, in cui ognuno inizia in una nuova pagina, con un titolo ed un numero progressivo di pagina per ogni file.

7.9 Alcune Funzioni

La libreria standard fornisce una varietà di funzioni, alcune delle quali si rivelano di particolare utilità. Abbiamo già menzionato le funzioni per il trattamento delle stringhe **strlen**, **strcpy**, **strcat** e **strcmp**. Eccone qui alcune altre.

Test e Conversioni sulle Classi di Caratteri

Parecchie macro effettuano test e conversioni di caratteri:

```
isalpha(c)      non-zero se c e' alfabetico, 0 se non lo e'.
isupper(c)      non-zero se c e' maiuscolo, 0 se non lo e'.
islower(c)      non-zero se c e' minuscolo, 0 se non lo e'.
isdigit(c)      non-zero se c e' una cifra, 0 se non lo e'.
isspace(c)      non-zero se c e' blank, tab o newline, 0 se non lo e'.
toupper(c)      converte c in maiuscolo
tolower(c)      converte c in minuscolo
```

Ungetc

La libreria standard fornisce una versione piuttosto restrittiva della funzione **ungetch** che abbiamo scritto nel Capitolo 4; è chiamata **ungetc**.

```
ungetc(c, fp)
```

rimanda indietro il carattere **c** nel file **fp**. Viene permesso solamente un carattere di rimando per file. **ungetc** può essere usata con qualsiasi funzione e macro di input come **scanf**, **getc** o **getchar**.

Chiamate al sistema

La funzione **system(s)** esegue il comando contenuto nella stringa di caratteri **s**, poi riprende l'esecuzione del programma corrente. I contenuti di **s** dipendono fortemente dal sistema operativo locale. Come esempio banale, su **UNIX**, la linea

```
system("date");
```

causa l'esecuzione del programma **date**; stampa la data e l'ora del giorno.

Gestione della Memoria

La funzione **calloc** è abbastanza simile alla **alloc** che abbiamo usato nei precedenti capitoli.

```
calloc(n, sizeof(oggetto))
```

ritorna un puntatore ad uno spazio sufficiente per **n** oggetti della grandezza specificata, oppure **NULL** se la richiesta non può essere soddisfatta. La memoria viene inizializzata a zero.
Il puntatore ha l'allineamento corretto per l'oggetto in questione, ma dev'essere forzato nel tipo appropriato, come in

```
char *calloc();
int *ip;

ip = (int *) calloc(n, sizeof(int));
```

cfree(p) rilascia lo spazio puntato da **p**, in cui **p** è stato originalmente ottenuto da una chiamata a **calloc**. Non ci sono restrizioni nell'ordine in cui viene rilasciato lo spazio, ma è un gravissimo errore rilasciare qualcosa che non sia stata ottenuta da una chiamata a **calloc**.
Il Capitolo 8 mostra l'implementazione di un allocatore di memoria come **calloc**, in cui i blocchi allocati possono essere rilasciati in qualunque ordine.

CAPITOLO 8

L'INTERFACCIA CON IL SISTEMA UNIX

Il tema di questo capitolo riguarda l'interfaccia tra i programmi C ed il sistema operativo **UNIX*** Poiché la maggior parte degli utenti C lavora con il sistema **UNIX**, dovrebbe essere utile alla maggioranza dei lettori. Anche se si usa il C su una macchina differente, studiando questi esempi si dovrebbe comunque riuscire e raggiungere una maggiore comprensione della programmazione C.
Il capitolo è diviso in tre parti principali: input/output, file system ed un allocatore di memoria. Le prime due parti presuppongono una modesta dimestichezza con le caratteristiche esterne di **UNIX**.
Nel capitolo 7 ci siamo occupati di un'interfaccia col sistema che si presenta uniforme su una varietà di sistemi operativi. Su ogni particolare sistema le routine della libreria standard devono essere scritte in termini delle possibilità di **I/O** effettivamente disponibili sul sistema. Nei primi paragrafi descriveremo i punti basilari disponibili per l'**I/O** sul sistema operativo **UNIX** e illustreremo come con essi si possano implementare parti della libreria standard.

8.1 Descrittori di file

Nel sistema operativo **UNIX** tutto l'input e l'output viene ottenuto leggendo o scrivendo files, poiché tutti i dispositivi periferici, compreso il terminale dell'utente, sono files del file system. Questo significa che una singola ed omogenea interfaccia tratta tutte le comunicazioni tra un programma e le periferiche.
Nel caso più generale, prima di leggere o scrivere un file, è necessario comunicare al sistema le proprie intenzioni con un procedimento chiamato "apertura" del file. Se si vuole scrivere su un file può essere necessario crearlo. Il sistema controlla i diritti sull'operazione (Il file esiste? Si hanno i permessi di accesso?), e, se tutto va bene, ritorna al programma un intero positivo piccolo chiamato "*descrittore di file*". Ogni volta che bisogna effettuare dell'**I/O** sul file, per identificare il file si usa il descrittore invece del nome. (Ciò è approssimativamente analogo all'uso di **READ(5,...)** e **WRITE(6,...)** in Fortran). Tutte le informazioni circa un file aperto vengono mantenute dal sistema; il programma dell'utente si riferisce al file solamente attraverso il descrittore di file.
Poiché l'input e output sul terminale dell'utente sono molto comuni, esistono speciali meccanismi per renderlo agevole. Quando l'interprete dei comandi (la "shell") esegue

---

* UNIX è un marchio registrato dei Bell Laboratories.

un programma, apre tre files con descrittori di file 0, 1 e 2 chiamati standard input, standard output e standard error output. Sono generalmente tutti connessi al terminale così se un programma legge il descrittore di file 0 e scrive sui descrittori di file 1 e 2, può ottenere l'I/O sul terminale senza preoccuparsi di aprire i files.
L'utente di un programma può *redirigere* l'**I/O** su e da files con < e >:

```
prog <infile >outfile
```

In questo caso la shell cambia gli assegnamenti di default per i descrittori di file 0 ed 1 dal terminale ai file in questione. Normalmente il descrittore di file 2 rimane collegato al terminale, così i messaggi di errore possono andare lì. Le stesse osservazioni sono valide se l'input o l'output è associato ad una pipe. Da notare che in tutti i casi gli assegnamenti dei file vengono cambiati dalla shell, non dal programma. Il programma non sa da dove arriva l'input né dove va l'output poiché usa il file 0 per l'input ed 1 e 2 per output.

8.2 I/O a basso livello — Read e Write

Il livello più basso di **I/O** in **UNIX** non fornisce alcun meccanismo di bufferizzazione né alcun altro servizio; è infatti un'entrata diretta nel sistema operativo. Tutto l'input e l'output viene effettuato da due funzioni chiamate **read** e **write**. Per entrambe, il primo argomento è un descrittore di file. Il secondo argomento è un buffer nel programma da cui provengono o in cui vanno i dati. Il terzo argomento è il numero di byte da trasferire. Le chiamate sono

```
n_letti = read(fd, buf, n);

n_scritti = write(fd, buf, n)
```

Ogni chiamata ritorna un conteggio di byte che è il numero di byte realmente trasferiti. In lettura, il numero di byte di ritorno può essere minore del numero richiesto. Un valore di ritorno uguale a zero byte implica la fine del file e, −1 indica un errore di qualche genere. In scrittura, il valore di ritorno è il numero di byte realmente scritti; è generalmente un errore se questo non è uguale al numero di byte che si suppone di aver scritto.
Il numero di byte da leggere o scrivere è abbastanza arbitrario. I due valori più comuni sono 1, che significa un carattere per volta ("senza buffer") e 512, che corrisponde all'ampiezza fisica di un blocco su molti dispositivi periferici. Questa ultima ampiezza sarà la più efficiente, ma non è eccessivamente dispendioso neanche l'I/O di un carattere per volta.
Collegando questi fattori, possiamo scrivere un semplice programma che copia l'input sull'output, l'equivalente del programma di copia di un file scritto nel Capitolo 1. In **UNIX**, questo programma copierà qualsiasi cosa su qualsiasi cosa, poiché l'input e l'output possono essere rediretti su ogni file o device.

```
#define BUFSIZE   512  /* ampiezza ottimale per lo UNIX
                          del PDP-11  */

main()    /* copia l'input sull'output  */
{
    char buf[BUFSIZE];
    int n;

    while ((n = read(0, buf, BUFSIZE)) > 0)
        write(1, buf, n);
}
```

Se l'ampiezza del file non è un multiplo di **BUFSIZE**, alcune **read** ritorneranno un numero di byte più piccolo da scrivere con **write**; la successiva chiamata a **read** dopo questa scrittura ritornerà zero.
È istruttivo vedere come **read** e **write** possano essere usate per costruire routine di livello più alto come **getchar**, **putchar**, ecc. Per esempio, questa è una versione di **getchar** che effettua input non bufferizzato.

```
#define   CMASK    0377  /* per avere i caratteri > 0  */

getchar()  /* input di un singolo carattere senza buffer */
{
    char c;

    return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

c *deve* essere dichiarato **char**, poiché **read** accetta un puntatore a carattere. Il carattere di ritorno dev'essere mascherato con **0377** per assicurarsi che sia positivo; altrimenti l'estensione del segno lo potrebbe rendere negativo. (La costante **0377** è appropriata per il **PDP-11** ma non necessariamente per altre macchine).
La seconda versione di **getchar** controlla l'input in grandi porzioni facendo uscire i caratteri uno per volta.

```
#define   CMASK  0377  /* per avere i caratteri > 0  */
#define   BUFSIZE  512

getchar()   /* versione con buffer  */
{
    static char  buf[BUFSIZE];
    static char  *bufp = buf;
    static int   n = 0;

    if (n == 0) { */* il buffer e' vuoto */
        n = read(0, buf, BUFSIZE);
        bufp = buf;
    }
    return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

8.3 Open, Creat, Close, Unlink

A parte i file predisposti standard input, output ed error, per leggere o scrivere files bisogna aprirli esplicitamente. Per fare questo ci sono due modi previsti dal sistema: **open** e **creat**.
open è abbastanza simile alla **fopen** trattata nel Capitolo 7 eccetto che invece di ritornare un puntatore al file, ritorna un descrittore di file, che non è altro che un **int**.

```
int fd;

fd = open(nome, modo-rw)
```

Come per **fopen**, l'argomento **nome** è una stringa di caratteri che corrisponde ad un nome esterno del file. L'argomento sulle modalità di accesso è differente: **modo-rw** è 0 in lettura, 1 in scrittura e 2 per gli accessi in lettura e scrittura. **open** ritorna −1 se viene fatto un qualsiasi errore; altrimenti ritorna un descrittore di file valido.
È un errore tentare di aprire tramite **open** un file che non esiste: è prevista la chiamata di sistema **creat** per creare nuovi files o per riscrivere i vecchi.

```
fd = creat(nome, modo-p)
```

ritorna un descrittore di file se è stata in grado di creare il file chiamato **nome**, in caso contrario ritorna −1. Se il file esiste già, **creat** lo ridurrà a lunghezza zero; per **creat** non è un errore il fatto che un file esista già.
Se il file è di un nuovo tipo, **creat** lo crea con la *modalità di protezione* specificata dall'argomento **modo-p**. Nel file system di **UNIX** ci sono nove bit di informazioni sulle protezioni associate ad un file, che controllano i permessi in lettura, scrittura ed esecuzione per il proprietario del file, per il gruppo di cui fa parte il proprietario e per tutti gli altri. Quindi un numero ottale di tre cifre è particolarmente adatto per specificare i permessi.
Per esempio, 0755 specifica il permesso di lettura, scrittura ed esecuzione per il proprietario, di lettura e di esecuzione per il gruppo e tutti gli altri.
Per chiarire, questa è una versione semplificata del programma di utilità **UNIX** *cp*, un programma che copia un file su un altro. (La principale semplificazione è che la nostra versione copia un solo file e non permette che il secondo argomento sia una directory).

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644  /* lettura-scrittura per il proprietario,
                       lettura per il gruppo e per gli altri */

main(argc, argv)     /* cp: copia f1 in f2  */
int argc;
char *argv[];
{
    int f1, f2, n;
    char buf[BUFSIZE];

    if (argc != 3)
        errore("Uso: cp da a", NULL);
    if ((f1 = open(argv[1], 0)) == -1)
        errore("cp: non posso aprire %s",argv[1]);
    if ((f2 = creat(argv[2], PMODE)) == -1)
        errore("cp: non posso creare %s", argv[2]);

    while ((n = read(f1, buf, BUFSIZE)) > 0)
        if (write(f2, buf, n) != n)
            errore("cp: errore di scrittura", NULL);
    exit(0);
}

errore(s1, s2)    /* stampa il messaggio di errore ed esce */
char *s1, *s2;
{
    printf(s1, s2);
    printf("\n");
    exit(1);
}
```

C'è un numero massimo di files (di solito 15-25) che possono essere aperti contemporaneamente in un programma. Di conseguenza, ogni programma che intende lavorare su molti files deve essere preparato a riutilizzare i descrittori di file. La routine **close** spezza la connessione tra un descrittore di file ed un file aperto e rilascia il descrittore di file per l'uso con qualche altro file. Il termine di un programma attraverso **exit** o come ritorno dal programma principale, chiude tutti i files aperti.
La funzione **unlink(nomefile)** rimuove il file **nomefile** dal file system.

Esercizio 8-1. Riscrivere il programma **cat** del Capitolo 7 usando **read**, **write**, **open** e **close** invece delle loro equivalenti nella libreria standard. Si facciano degli esperimenti per determinare le velocità relative delle due versioni.

8.4 Accesso Casuale — Seek e Lseek

L'**I/O** su file è generalmente sequenziale: ogni **read** o **write** avviene nella posizione del file successiva alla precedente operazione. Quando è necessario, comunque, un file può essere letto o scritto in ordine arbitrario. La chiamata di sistema **lseek** fornisce un modo per muoversi all'interno di un file senza dover di fatto leggere o scrivere:

```
lseek(fd, distanza, origine);
```

forza la posizione corrente nel file il cui descrittore è **fd** a spostarsi verso la posizione **distanza**, che è relativa alla posizione specificata da **origine**. Letture o scritture successive inizieranno in quella posizione. **distanza** è un **long**. **fd** e **origine** sono **int**. **origine** può essere 0, 1 o 2 per specificare se lo scarto debba essere misurato rispettivamente dall'inizio, dalla posizione corrente o dalla fine del file. Per esempio, per appendere in coda ad un file ci si posiziona alla fine prima di scrivere:

```
lseek(fd, 0L, 2);
```

Per ritornare all'inizio ("rewind")

```
lseek(fd, 0L, 0);
```

Da notare l'argomento **0L**; potrebbe anche essere scritto come **(long) 0**.
Con **lseek** è possibile trattare i files più o meno come grandi array al prezzo di un accesso più lento. Per esempio, la semplice seguente funzione legge un numero qualsiasi di byte da qualsiasi posizione arbitraria all'interno di un file.

```
get(fd, pos, buf, n)  /* legge n byte dalla posizione pos */
int fd, n;
long pos;
char *buf;
{
    lseek(fd, pos, 0);   /* va a pos  */
    return(read(fd, buf, n));
}
```

Prima della versione 7 di **UNIX**, il meccanismo basilare di accesso al sistema di I/O era chiamato **seek**. **seek** è identica a **lseek** a parte il fatto che il suo argomento **distanza** è un **int** invece che un **long**. Di conseguenza, poiché gli interi del **PDP-11** hanno solamente 16 bit, la **distanza** specificata a **seek** è limitato a 65535; per questa ragione i valori di **origine** uguali a 3, 4, 5 fanno in modo che **seek** moltiplichi lo scarto dato per 512 (il numero di byte di un blocco fisico) e poi interpreti **origine** come se fosse rispettivamente 0, 1 o 2. Quindi per posizionarsi arbitrariamente all'interno di un grosso file sono necessarie due **seek**, la prima che seleziona il blocco e l'altra che ha **origine** uguale a 1 e si sposta sul byte desiderato all'interno del blocco.

Esercizio 8-2. Chiaramente, **seek** può essere scritta in termini di **lseek** e viceversa. Si riscriva ognuna in termini dell'altra.

8.5 Esempio — Un'implementazione di Fopen e Getc

Vediamo ora come si raggruppano alcuni di questi frammenti illustrando un'implementazione delle routine della libreria standard **fopen** e **getc**.
Si ricordi che i files sono descritti nella libreria standard in termini di puntatori a file invece che descrittori di file. Un puntatore a file è un puntatore ad una struttura che contiene parecchie informazioni circa il file: un puntatore al buffer in modo che il file possa essere letto a grandi pezzi; un contatore del numero dei caratteri lasciati nel buffer;

un puntatore alla posizione del carattere successivo nel buffer; alcune segnalazioni che descrivono le modalità di lettura/scrittura, ecc.; ed il descrittore del file.
La struttura dati che descrive un file è contenuta nel file **stdio.h**, che deve essere incluso (con **#include**) in ogni file sorgente che usa le routine della libreria standard. Viene anche incluso dalle funzioni di quella stessa libreria. Nella seguente citazione da **stdio.h**, i nomi che vengono usati solamente dalle funzioni della librerai iniziano con un underscore in modo che siano più difficilmente simili ai nomi di un programma dell'utente.

```
#define     _BUFSIZE     512
#define     _NFILE        20

typedef     struct     _iobuf {
    char     *_ptr;     /* posizione del carattere successivo */
    int     _cnt;       /* numero di caratteri lasciati */
    char     *_base;    /* posizione del buffer */
    char     _flag;     /* modalita' di accesso al file */
    char     _fd;       /* descrittore del file */
} FILE;
extern FILE _iob[_NFILE];

#define     stdin     (&_iob[0])
#define     stdout     (&_iob[1])
#define     stderr     (&_iob[2])

#define    _READ      01    /* apertura del file in lettura */
#define    _WRITE     02    /* apertura del file in scrittura */
#define    _UNBUF     04    /* il file e' senza buffer */
#define    _BIGBUF    010   /* allocato un grosso buffer */
#define    _EOF   020       /* in questo file e' stato incontrato EOF */
#define    _ERR   040       /* in questo file e' stato incontrato
                               un errore */
#define    NULL   0
#define    EOF    (-1)

#define     getc(p)      (--(p)->_cnt >= 0 \
                 ? *(p)->_ptr++ & 0377 : _fillbuf(p))
#define     getchar()    getc(stdin)

#define     putc(x,p)    (--(p)->_cnt >= 0 \
                 ? *(p)->_ptr++ = (x) : _flushbuf((x),p))
#define     putchar(x)   putc(x,stdout)
```

Normalmente la macro **getc** diminuisce solo il contatore, avanza il puntatore, e ritorna il carattere. (Una **#define** lunga può continuare sulla linea successiva con una backslash). Se il contatore diventa negativo, **getc** chiama la funzione **__fillbuf** per riempire il buffer, rinizializzare il contenuto della struttura e ritornare un carattere. Una funzione può presentare una interfaccia portabile anche se contiene essa stessa costrutti non portabili: **getc** maschera i caratteri con **0377**, che annulla l'estensione di segno effettuata dal **PDP-11** e assicura che tutti i caratteri saranno positivi.
Anche se non entreremo nei dettagli, abbiamo incluso la definizione di **putc** per mostrare che si comporta in modo molto simile a **getc**, chiamando la funzione **__flushbuf** quando il buffer è pieno.
Si può ora scrivere la funzione **fopen**. La maggior parte di **fopen** consiste nell'aprire il

file, posizionarsi al punto giusto e disporre i bit di segnalazione per indicare lo stato appropriato. **fopen** non alloca nessuno spazio per il buffer; ciò viene effettuato da **_fillbuf** alla prima lettura.

```
#include <stdio.h>
#define  PMODE  0644   /* lettura e scrittura per il proprietario,
                          lettura per gli altri  */

FILE *fopen(name, mode)  /* apre file, ritorna il puntatore al file  */
register char *name, *mode;
{
    register int fd;
    regster  FILE *fp;

    if (*mode != 'r' && *mode != 'w' && *mode != 'a')  {
        fprintf(stderr, "%s : modo scorretto per aprire %s\n",
            mode, name);
        exit(1);
    }
    for (fp = _iob; fp < _iob + _NFILE; fp++)
        if ((fp->_flag & (_READ | _WRITE)) == 0)
            break;   /* e' stato trovato un posto libero */
    if (fp >= _iob + _NFILE)  /* nessun posto libero */
        return(NULL);

    if (*mode == 'w')   /* accesso al file */
        fd = creat(name, PMODE);
    else if (*mode == 'a')  {
        if ((fd = open(name, 1)) == -1)
            fd = creat(name, PMODE);
        lseek(fd, 0L, 2);
    }
    else
        fd = open(name, 0);
    if (fd == -1)   /* non si puo' accedere a name */
        return(NULL);

    fp->_fd = fd;
    fp->_cnt = 0;
    fp->_base = NULL;
    fp->_flag &= ~(_READ | WRITE);
    fp->_flag |= (*mode == 'r') ? _READ : _WRITE;
    return(fp);
}
```

La funzione **_fillbuf** è parecchio più complicata. La maggiore complessità sta nel fatto che **_fillbuf** tenta di permettere l'accesso al file anche se non c'è abbastanza memoria per bufferizzare l'I/O. Se da **calloc** si può ottenere spazio per un nuovo buffer, va tutto bene; altrimenti **_fillbuf** ottiene l'**I/O** non bufferizzato usando un singolo carattere posto in un array privato.

```
#include <stdio.h>

_fillbuf(fp)  /* alloca e carica il buffer di input */
register FILE *fp;
{
    static char smallbuf[_NFILE];  /* per I/O senza buffer */
    char *calloc();

    if ((fp->_flag&_READ)==0 || (fp->_flag&(_EOF|_ERR)) != 0)
        return(EOF);
    while (fp->_base == NULL)  /* trovato lo spazio per il buffer */
        if (fp->_flag & _UNBUF)  /* senza buffer */
            fp->_base = &smallbuf[fp->_fd];
        else if ((fp->_base=calloc(_BUFSIZE, 1)) == NULL)
            fp->_flag |= _UNBUF; /* non si puo' avere un grande buffer */
        else
            fp->_flag |= _BIGBUF;  /* se ne ha uno grande */
    fp->_ptr = fp->_base;
    fp->_cnt = read(fp->_fd, fp->_ptr,
                    fp->_flag & _UNBUF ? 1 : _BUFSIZE);
    if (--fp->_cnt < 0) {
        if (fp->_cnt == -1)
            fp->_flag |= _EOF;
        else
            fp->_flag |= _ERR;
        fp->_cnt = 0;
        return(EOF);
    }
    return(*fp->_ptr++ & 0377);  /* rende il carattere positivo */
}
```

La prima chiamata a **getc** per un certo file trova il contatore a zero, il che forza una chiamata a **_fillbuf**. Se **_fillbuf** trova che il file non è stato aperto in lettura, ritorna immediatamente **EOF**. Altrimenti, tenta di allocare un buffer grande e, se fallisce, un buffer di un singolo carattere, definendo appropriatamente le informazioni per bufferizzare in **_flag**.
Dopo che è stato stabilito il buffer, **_fillbuf** chiama semplicemente **read** per riempirlo, inizializza il conteggio ed i puntatori e ritorna il carattere all'inizio del buffer. Successive chiamate a **_fillbuf** troveranno allocato un buffer.
Ora l'unica cosa che rimane è come inizia il tutto. L'array **_iob** dev'essere definito e inizializzato per **stdin**, **stdout** e **stderr**:

```
FILE _iob[_NFILE] ={
    { NULL, 0, NULL, _READ, 0 },    /* stdin */
    { NULL, 0, NULL, _WRITE, 1 },   /* stdout */
    { NULL, 0, NULL, _WRITE | _UNBUF, 2 }  /* stderr */
};
```

L'inizializzazione della parte **_flag** della struttura mostra che **stdin** è in lettura, **stdout** è in scrittura e **stderr** è in scrittura senza buffer.

Esercizio 8-3. Riscrivere **fopen** e **_fillbuf** con i campi invece di esplicite operazioni sui bit.

Esercizio 8-4. Strutturare e scrivere le routine **_flushbuf** e **fclose**.

Esercizio 8-5. La libreria standard fornisce la funzione

```
fseek(fp, scarto, origine)
```

che è identica a **lseek** eccetto che **fp** è un puntatore a file invece che un descrittore di file. Scrivere **fseek**. Ci si assicuri che la propria **fseek** si coordini correttamente con le operazioni sul buffer delle altre funzioni di libreria.

8.6 Esempio — Lista di Directory

A volte è necessario un tipo diverso di interazione col file system per determinare informazioni *circa* un file, non sul suo contenuto. Il comando di **UNIX** *ls* ("lista directory") è un esempio: stampa i nomi dei files in una directory e, opzionalmente, altre informazioni come le dimensioni, permessi e così via.
Poiché sotto **UNIX** una directory non è altro che un file, non c'è niente di particolare non è altro che file, non c'è niente di particolare su un comando come *ls*; legge un file e estrae le parti rilevanti delle informazioni trovate. In ogni caso, il formato di tali informazioni è determinato dal sistema, non da un programma utente, quindi ls deve sapere come il sistema le rappresenta.
Illustreremo alcuni di questi aspetti scrivendo un programma chiamato *fsize*. *fsize* è una particolare forma di *ls* che stampa le dimensioni di tutti i files nominati nell'elenco di argomenti. Se uno dei file è una directory, *fsize* chiama se stesso ricorsivamente su quella directory. Se non c'è alcun argomento, esamina la directory corrente.
Per iniziare, ripassiamo velocemente la struttura del file system. Una directory è un file che contiene un elenco di nomi di files ed alcune indicazioni sulla loro locazione. In effetti la "locazione" è un indice ad un'altra tavola chiamata "tavola degli inode". L'inode di un file è il luogo in cui si mantengono tutte le informazioni circa un file a parte il suo nome. Un elemento di una directory consiste solamente di due oggetti: un numero di inode ed il nome del file. La definizione precisa si ha includendo il file **sys/dir.h** che contiene

```
#define DIRSIZ  14   /* lunghezza massima di un nome di file */

struct  direct   /* struttura di un elemento di una directory */
{
    ino_t   d_ino;   /* numero di inode */
    char    d_name[DIRSIZ];  /* nome del file */
};
```

Il "tipo" **ino_t** è una **typedef** che descrive l'indice all'interno della tavola di inode. Nello **UNIX** del **PDP-11** è **unsigned** ma non è questo il tipo di informazione da conglobare in un programma: potrebbe essere diverso per altri sistemi. Questo il motivo della **typedef**. In **sys/types.h** si trova un completo insieme di tipi di "sistema".
La funzione **stat** prende un nome di file e ritorna tutte le informazioni nell'inode per quel file (o — 1 se c'è un errore). Cioè,

```
struct stat stbuf;
char *name;

stat(name, &stbuf);
```

riempie la struttura **stbuf** con le informazioni nell'inode del file **name**. La struttura che descrive il valore ritornato da **stat** è in **sys/stat.h** ed appare così:

```
struct  stat  /* struttura ritornata da stat */
{
    dev_t    st_dev;    /* device dell'inode */
    ino_t    st_ino;    /* numero dell'inode */
    short    st_mode;   /* bit di modalita' di accesso */
    short    st_nlink;  /* numero di link al file */
    short    st_uid;    /* userid del proprietario */
    short    st_gid;    /* gruppo del proprietario */
    dev_t    st_rdev;   /* per i file speciali */
    off_t    st_size;   /* ampiezza del file in caratteri */
    time_t   st_atime;  /* tempo di ultimo accesso */
    time_t   st_mtime;  /* tempo di ultima modifica */
    time_t   st_ctime;  /* tempo di prima creazione  */
};
```

Molti campi sono già spiegati dai commenti. **st_mode** contiene un insieme di segnalazioni che descrivono il file; per comodità, anche queste definizioni sono parte del file **sys/stat.h**

```
#define S_IFMT     0160000    /* tipo del file */
#define S_IFDIR    0040000    /* directory */
#define S_IFCHR    0020000    /* character special */
#define S_IFBLK    0060000    /* block special */
#define S_IFREG    0100000    /* normale */
#define S_ISUID    04000      /* set user id on execution */
#define S_ISGID    02000      /* set group id on execution */
#define S_ISVTX    01000      /* mantiene l'immagine nell'area di
                                 swap anche dopo l'uso */
#define S_IREAD    0400       /* permesso in lettura, proprietario */
#define S_IWRITE   0200       /* permesso in scrittura, proprietario */
#define S_IEXEC    0100       /* perm. in ricerca/esecuzione,
                                 proprietario */
```

Ora siamo in grado di scrivere il programma *fsize*. Se la modalità indicata da **stat** indica che il file non è una directory, allora si ha a disposizione la sua dimensione e si può stampare direttamente. Se è una directory, invece, bisogna analizzare quella directory file per file; di volta in volta si possono incontrare sottodirectory, così il processo diventa ricorsivo.
La routine principale come al solito tratta principalmente con gli argomenti sulla linea di comando; passa ogni argomento a *fsize* in un grosso buffer.

```
#include <stdio.h>
#include <sys/types.h>    /* typedef */
#include <sys/dir.h>      /* struttura della directory */
#include <sys/stat.h>     /* struttura ritornata di stat */
#define     BUFSIZE   256

main(argc, argv)   /* fsize: stampa le dimensioni dei files */
char *argv[];
{
    char buf[BUFSIZE];

    if (argc == 1) {  /* default: directory corrente */
        strcpy(buf, ".");
        fsize(buf);
    } else
        while (--argc > 0) {
            strcpy(buf, *++argv);
            fsize(buf);
        }
}
```

La funzione *fsize* stampa la dimensione del file. Se il file è una directory **fsize** chiama prima **directory** per raggiungere tutti i file in essa contenuti. Da notare l'uso dei nomi delle segnalazioni **S_IFMT** e **S_IFDIR** da **stat.h**.

```
fsize(name)    /* stampa la grandezza di name */
char *name;
{
    struct stat stbuf;

    if (stat(name, &stbuf) == -1)  {
        fprintf(stderr, "fsize: non trovo %s\n", name);
        return;
    }
    if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
        directory(name);
    printf("%8ld %s\n", stbuf.st_size, name);
}
```

La funzione **directory** è la più complicata. Gran parte di essa, comunque, si adopera a creare il pathname completo del file che sta trattando.

```
directory(name)    /* fsize per tutti i file in name */
char *name;
{
    struct direct dirbuf;
    char *nbp, *nep;
    int i, fd;

    nbp = name + strlen(name);
    *nbp++ = '/';    /* aggiunge lo slash al nome della directory */
    if (nbp+DIRSIZ+2 >= name+BUFSIZE) /* nome troppo lungo */
        return;
    while (read(fd, (char *)&dirbuf, sizeof(dirbuf)) >= 0) {
        if (dirbuf.d_ino == 0)    /* posto non usato */
            continue;
        if (strcmp(dirbuf.d_name, ".") == 0
            || strcmp(dirbuf.d_name, "..") == 0)
            continue;  /* salta se stessa e la directory superiore */
        for (i=0; nep=nbp; i < DIRSIZ; i++)
            *nep++ = dirbuf.d_name[i];
        *nep++ = '\0';
        fsize(name);
    }
    close(fd);
    *--nbp = '\0';  /* rimette il nome */
}
```

Se in una directory c'è un posto inutilizzato (perché è stato rimosso un file), la parte dell'inode è zero e la posizione viene saltata. Ogni directory contiene anche il posto per se stessa, chiamata ".", e della sua directory padre, "..";
Chiaramente anche queste ultime devono essere saltate altrimenti il programma resterà in esecuzione molto a lungo.
Anche se il programma *fsize* è piuttosto specializzato, indica un paio di concetti importanti. Primo, molti programmi non sono "programmi di sistema"; essi usano semplicemente le informazioni la cui forma o contenuto vengono acquisite dal sistema operativo. Secondo, per alcuni programmi è determinante che la rappresentazione delle informazioni appaia solamente in "file di testa" standard come **stat.h** e **dir.h** e che i programmi includano tali file invece di conglobare le dichiarazioni all'interno di essi stessi.

8.7 Esempio — Un allocatore di memoria

Nel Capitolo 5 abbiamo visto una semplice versione di **alloc**. La versione che scriviamo ora non è limitata: le chiamate ad **alloc** e **free** possono essere scambiate in qualsiasi ordine; **alloc** chiama il sistema operativo per ottenere la memoria necessaria. Per poter essere utili nei propri propositi, queste routines richiamano alcune considerazioni circa lo scrivere codice dipendente dalla macchina in un modo relativamente indipendente dalla macchina e mostrano inoltre un'applicazione pratica delle strutture, unioni e **typedef**.
Invece di allocare in un'array di ampiezza fissa ottenuto in compilazione, **alloc** richiede lo spazio necessario al sistema operativo. Poiché nel programma altre attività potrebbero richiedere spazio in modo asincrono, lo spazio che gestisce **alloc** non dev'essere

contiguo. Quindi la sua memoria libera viene mantenuta come una catena di blocchi liberi. Ogni blocco contiene un'ampiezza, un puntatore al successivo blocco e allo spazio stesso. I blocchi sono mantenuti in ordine crescente di indirizzo di memoria e l'ultimo blocco (con indirizzo più alto) punta al primo così la catena è un anello vero e proprio.
Quando si effettua una richiesta, viene scandita la lista dei blocchi liberi fino a che se ne trova uno sufficientemente grande. Se il blocco è esattamente dell'ampiezza richiesta, viene scollegato dalla lista e ritorna all'utente. Se il blocco è troppo grande viene spezzato ritornando la esatta quantità all'utente mentre il rimanente viene rimandato nella lista dei blocchi liberi. Se non si trova alcun blocco sufficientemente grande, viene ottenuto un altro bloco dal sistema operativo collegandolo alla lista; poi si riprende la ricerca.
Anche il liberare la memoria causa una ricerca nella lista per trovare il posto adatto per inserire il blocco appena rilasciato. Se il blocco da liberare è adiacente a un blocco nella lista in qualsiasi delle due parti, vengono fusi insieme formando un unico blocco più grande, così la memoria non diventa troppo frammentata. È facile determinare l'adiacenza poiché la lista viene mantenuta in ordine di memoria.
Un problema, accennato nel Capitolo 5, è di assicurarsi che la memoria ritornata da **alloc** sia correttamente allineata per gli oggetti che verranno allocati all'interno di essa. Anche se le macchine sono differenti, per ogni macchina c'è un tipo più restrittivo: se il tipo più restrittivo può essere allocato ad un particolare indirizzo, lo possono essere anche tutti gli altri tipi. Per esempio, su **IBM** 360/370, Honeywell 6000 e parecchie altre macchine, qualsiasi oggetto può essere allocato ad un allineamento appropriato per un **double**; sul **PDP-11** è sufficiente che lo sia per un **int**.
Un blocco libero contiene un puntatore al blocco successivo nella catena, una registrazione dell'ampiezza del blocco e poi lo spazio libero stesso; le informazioni di controllo all'inizio sono chiamate "header". Per semplificare l'allineamento, tutti i blocchi sono multipli dell'ampiezza dello header ed esso stesso è opportunamente allineato. Questo si ottiene per mezzo di una union che contiene la struttura header desiderata ed un'istanza del tipo con l'allineamento più restrittivo:

```
typedef int ALIGN;  /* forza l'allineamento sul PDP-11 */

union header {  /* informazioni di testa del blocco libero */
    struct {
        union header *ptr;  /* successivo blocco libero */
        unsigned size; /* ampiezza di questo blocco libero */
    } s;
    ALIGN    x; /* forza l'allineamento dei blocchi */
};

typedef union header HEADER;
```

In **alloc**, la quantità richiesta in caratteri viene arrotondata al numero appropriato di unità di ampiezza dello header; il blocco che sarà allocato conterrà una unità in più per l'header stesso, e questo sarà il valore registrato nel campo *size* dello header. Il puntatore ritornato da **alloc** punta allo spazio libero, non allo header.

```
static HEADER base  /* lista vuota per iniziare */
static HEADER *allocp = NULL;  /* ultimo blocco allocato */

char *alloc(nbytes)  /* allocatore di memoria di generale utilita' */
unsigned nbytes;
{
    HEADER *morecore();
    register HEADER *p, *q;
    register int nunits;

    nunits = 1+(nbytes+sizeof(HEADER)-1)/sizeof(HEADER);
    if ((q = allocp) == NULL) {  /* non c'e' ancora lista dei
                                    blocchi liberi */
        base.s.ptr = allocp = q = &base;
        base.s.size = 0;
    }
    for (p=q->s.ptr; ; q=p, p=p->s.ptr) {
        if (p->s.size >= nunits) {   /* grande a sufficienza */
            if (p->s.size == nunits)  /* esatto */
                q->s.ptr = p->s.ptr;
            else {   /* alloca la parte finale */
                p->s.size -= nunits;
                p+= p->s.size;
                p->s.size = nunits;
            }
            allocp = q;
            return((char *)(p+1));
        }
        if (p == allocp)  /* avvolto intorno alla lista libera */
            if ((p = morecore(nunits)) == NULL)
                return(NULL);  /* non ne e' piu' rimasta */
    }
}
```

La variabile **base** viene usata per iniziare; se **allocp** è **NULL**, come lo è alla prima chiamata di **alloc**, allora viene creata una lista degenerata: contiene un blocco di ampiezza zero e punta a se stesso. In ogni caso, la lista viene scandita. La ricerca di un blocco libero di ampiezza adeguata inizia al punto **(allocp)** in cui era stato trovato l'ultimo blocco; questo metodo aiuta a mantenere la lista omogenea. Se si trova un blocco troppo grande, all'utente viene ritornata la parte finale; in questo modo lo header originale deve solo modificare la propria ampiezza. In ogni caso il puntatore che viene ritornato all'utente è all'area libera, che si trova un'unità oltre lo header. Da notare che **p** viene convertito ad un puntatore carattere prima di essere restituito da **alloc**.
La funzione **morecore** ottiene la memoria dal sistema operativo. I dettagli su come ciò viene ottenuto variano naturalmente da sistema a sistema. In **UNIX**, la chiamata di sistema **sbrk(n)** ritorna un puntatore ad **n** ulteriori byte di memoria. (Il puntatore soddisfa tutte le restrizioni di allineamento).
Poiché chiedere la memoria al sistema è un'operazione relativamente pesante non vogliamo eseguirla ad ogni chiamata ad **alloc**, così **morecore** arrotonda il numero di unità richieste in un valore più grande; questo grande blocco sarà spezzato a seconda

delle richieste. Il valore di arrotondamento è un parametro che può essere regolato a piacere.

```
#define NALLOC  128   /* n. unita' da allocare in una volta sola */

static HEADER *morecore(nu)  /* chiede la memoria al sistema */
unsigned nu;
{
    char *sbrk();
    register char *cp;
    register HEADER *up;
    register int rnu;

    rnu = NALLOC * ((nu+NALLOC-1) / NALLOC);
    cp = sbrk(rnu * sizeof(HEADER));
    if ((int)cp == -1)   /* non c'e' alcuno spazio */
        return(NULL);
    up = (HEADER *)cp;
    up->s.size = rnu;
    free((char *)(up+1));
    return(allocp);
}
```

sbrk ritorna −1 se non c'era spazio, anche se **NULL** sarebbe stato una scelta migliore. −1 dev'essere convertito ad un **int** così può essere confrontato senza problemi. Oltre a ciò i cast sono pesantemente usati affinché la funzione sia relativamente indipendente dai dettagli di rappresentazione dei puntatori su differenti macchine.
L'ultima cosa è **free**. Scorre semplicemente la lista iniziando ad **allocp** cercando il posto per inserire il blocco libero, questo sia tra due blocchi esistenti che a una delle due estremità. In ogni caso, se il blocco che si sta liberando è adiacente ad uno dei due vicini, i blocchi adiacenti vengono combinati. L'unico problema è quello di mantenere puntatori che puntino agli oggetti giusti e le dimensioni corrette.

```
free(ap)   /* manda il blocco ap nella lista libera */
char *ap;
{
    register HEADER *p, *q;

    p = (HEADER *)ap -1;   /* punta a header */
    for (q=allocp; !(p > q && p < q->s.ptr); q=q->s.ptr)
        if (q >= q->s.ptr && (p > q || p < q->s.ptr))
            break;   /* a una o all'altra fine */

    if (p+p->s.size == q->s.ptr) {
    /* si attacca al blocco piu' in alto */
        p->s.size += q->s.ptr->s.size;
        p->s.ptr = q->s.ptr->s.ptr;
    } else
        p->s.ptr = q->s.ptr;
    if (q+q->s.size == p) {
    /* si attacca al blocco piu' in basso */
        q->s.size += p->s.size;
        q->s.ptr = p->s.ptr;
    } else
        q->s.ptr = p;
    allocp = q;
}
```

Sebbene l'allocazione di memoria sia intrinsicamente dipendente dalla macchina, il codice appena scritto mostra come le dipendenze possono essere confinate in una parte molto piccola del programma. L'uso di **typedef** e **union** controlla l'allineamento (a condizione che **sbrk** fornisca un puntatore appropriato). I cast fanno in modo che le conversioni dei puntatori siano ottenute esplicitamente rimediando persino ad un'interfaccia col sistema mal progettata. Anche se questi dettagli sono relativi all'allocazione di memoria, questo tipo di approccio generale è ugualmente applicabile ad altre situazioni.

Esercizio 8-6. La funzione della libreria standard **calloc(n, size)** ritorna un puntatore ad **n** oggetti di ampiezza **size**, con la memoria inizializzata a zero. Scrivere **calloc**, usando **alloc** come modello o come funzione che viene chiamata.

Esercizio 8-7. alloc accetta una richiesta di ampiezza senza controllare la sua plausibilità; **free** crede che il blocco che deve liberare contenga un valido campo di ampiezza. Implementare queste routine in modo che abbiano più cura dei casi d'errore.

Esercizio 8-8. Scrivere la routine **bfree(p, n)** che libera un arbitrario blocco **p** di **n** caratteri all'interno della lista mantenuta da **alloc** e **free**. Usando **bfree**, un utente può in ogni momento aggiungere un array statico o esterno alla free list.

APPENDICE A

MANUALE DI RIFERIMENTO DEL C

1. Introduzione

Questo manuale descrive il linguaggio C del DEC PDP-11, dell'Honeywell 6000, dell'IBM System/370 e dell'Interdata 8/32. Dove vi sono differenze, ci si concentra sul PDP-11, pur cercando di far notare i dettagli dipendenti dalla implementazione. A parte poche eccezioni, queste dipendenze seguono direttamente le caratteristiche di base dell'hardware; i vari compilatori sono generalmente piuttosto compatibili.

2. Convenzioni Lessicali

Ci sono sei classi di simboli: identificatori, parole chiave, costanti, stringhe, operatori ed altri separatori. I blank, tab, newline ed i commenti (nell'insieme, "spazi bianchi"), come sotto descritto vengono ignorati a parte che servono per separare i simboli. Qualche spazio bianco è necessario per separare identificatori, parole chiave e costanti che sarebbero altrimenti adiacenti.
Se il flusso di input è stato suddiviso in simboli fino a un dato carattere, si assume che il simbolo successivo sia la più lunga stringa di caratteri che potrebbe eventualmente costituire un simbolo.

2.1 Commenti

I caratteri /* introducono un commento, che termina con i caratteri */. I commenti non possono essere nidificati.

2.2 Identificatori (Nomi)

Un identificatore è una sequenza di lettere e cifre; il primo carattere dev'essere una lettera. L'underscore _ è considerato una lettera. Le lettere maiuscole e minuscole sono differenti. Sono significativi soltanto i primi otto caratteri, sebbene sia possibile usarne di più. Gli identificatori esterni, che vengono usati dai vari assemblatori e caricatori, sono più ristretti:

DEC PDP/11	7 caratteri, 2 forme
Honeywell 6000	6 caratteri, 1 forma
IBM 360/370	7 caratteri, 1 forma
Interdata 8/32	8 caratteri, 2 forme

2.3 Parole Chiave

I seguenti identificatori sono riservati all'uso come parole chiave e non possono essere utilizzati in altro modo:

```
int          extern       else
char         register     for
float        typedef      do
double       static       while
struct       goto         switch
union        return       case
long         sizeof       default
short        break        entry
unsigned     continue
auto         if
```

La parola chiave **entry** non è ancora implementata da nessun compilatore ma è destinata ad un uso successivo. Alcune implementazioni riservano anche le parole chiave **fortran** e **asm**.

2.4. Costanti

Ci sono vari tipi di costanti, che sono elencati più avanti. Le caratteristiche hardware che riguardano le dimensioni sono riassunte nel paragrafo 2.6.

2.4.1. Costanti Intere

Una costante intera è composta da una sequenza di cifre; è ottale se inizia con 0 (cifra zero), è in caso contrario decimale. Le cifre 8 e 9 hanno rispettivamente valore ottale 10 e 11. Una sequenza di cifre precedute da **0x** o **0X** (cifra zero) rappresenta un intero esadecimale. Le cifre esadecimali includono i caratteri da **a** o **A** fino a **f** o **F** con valori da 10 a 15. Una costante decimale il cui valore supera il più grosso intero con segno della macchina, viene considerata **long**; una costante ottale o esadecimale che supera il più grosso intero senza segno della macchina, viene ugualmente assunta essere **long**.

2.4.2 Costanti long esplicite

Una costante intera decimale, ottale o esadecimale immediatamente seguita da **l** (lettera elle) o **L** è una costante long. Come abbiamo già visto, su qualche macchina i valori interi e long possono essere considerati identici.

2.4.3 Costanti Carattere

Una costante carattere è un carattere racchiuso tra singoli apici, come in **'x'**. Il valore di una costante carattere è il valore numerico del carattere nel set di caratteri della macchina.

Alcuni caratteri non grafici, l'apice singolo ' e backslash \ possono essere rappresentati secondo le sequenze della seguente tabella:

```
newline            NL(LF)     \n
tab orizzontale    HT         \t
backspace          BS         \b
carriage return    CR         \r
form feed          FF         \f
backslash          \          \\
apice singolo      '          \'
modello a bit      ddd        \ddd
```

L'escape *ddd* consiste di backslash seguito da 1, 2 o 3 cifre ottali che specificano il valore del carattere desiderato. Un caso particolare di questa costruzione è **\0** (non è seguito da cifre), che rappresenta il carattere **NULL**. Se il carattere che segue un backslash non è compreso tra i caratteri specificati, il backslash viene ignorato.

2.4.4 Costanti in Virgola Mobile

Una costante floating si compone di una parte intera, un punto decimale, una parte decimale, un **e** o **E** e facoltativamente di un esponente intero con segno. La parte intera e quella decimale consistono entrambe di una sequenza di cifre. Può essere omessa sia la parte intera o la parte decimale, ma non entrambe; può essere omesso il punto decimale o l'e e l'esponente, ma non entrambi. Ogni costante floating è considerata in doppia precisione.

2.5 Stringhe

Una stringa è una sequenza di caratteri delimitati da doppi apici, come in "...". Una stringa ha tipo "array di caratteri" e classe di allocazione **static** (vedi paragrafo 4 più avanti) e viene inizializzata con i caratteri ricevuti. Tutte le stringhe, anche quando sono scritte identiche, sono diverse. Il compilatore immette un byte nullo **\0** alla fine di ogni stringa in maniera che i programmi che esaminano la stringa ne possano trovare la fine. In una stringa, il carattere doppio apice "deve essere proceduto da \; inoltre si possono usare le stesse sequenze descritte per le costanti carattere. Infine, un \ e un newline immediatamente seguente vengono ignorati.

2.6 Caratteristiche Hardware

La seguente tabella riassume certe proprietà hardware che variano da macchina a macchina. Pur influendo sulla portabilità del programma, rappresentano un problema minore di quanto si possa pensare *a priori*.

	DEC PDP/ 11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	EBCDIC	ASCII
char	8 bits	9 bits	8 bits	8 bits
int	16	36	32	32
short	16	36	16	16
long	32	36	32	32
float	32	36	32	32
double	64	72	64	64
range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 76}$

Per queste quattro macchine, i numeri in floating point hanno esponenti di 8 bit.

3. Notazione Sintattica

Nella notazione sintattica usata in questo manuale, le categorie sintattiche sono indicate dal carattere tipografico *corsivo*, ed i caratteri e le parole da prendere alla lettera in **neretto**. Categorie alternative sono elencate in righe separate. Un simbolo opzionale terminale o non-terminale viene indicato dal pedice "opz" scritto sotto, per cui

$$\{espressione_{opz}\}$$

indica un'espressione opzionale racchiusa tra parentesi graffe. La sintassi è riassunta nel paragrafo 18.

4. Cosa c'è in un nome?

Il C basa l'interpretazione di un identificatore su due attributi dell'identificatore stesso: la sua *classe di memorizzazione* e il suo *tipo*. La classe di memorizzazione determina l'allocazione e la durata della memoria associata ad un identificatore; il tipo determina il significato dei valori trovati nella parte di memoria riservata all'identificatore.

Ci sono quattro classi dichiarabili di allocazione: automatiche, statiche, esterne e registri. Le variabili automatiche sono relative ad ogni chiamata del blocco (paragrafo 9.2), e vengono abbandonate all'uscita dello stesso; le variabili static sono relative a un blocco, ma mantengono il loro valore al rientro del blocco anche dopo che il controllo lo ha abbandonato. Le variabili esterne esistono e trattengono i loro valori per tutta l'esecuzione dell'intero programma, e possono essere usate per le comunicazioni tra funzioni, anche se compilate separatamente. Le variabili registri sono, se possibile, allocate nei veloci registri della macchina; come le variabili automatiche, esse sono locali ad ogni blocco e scompaiono all'uscita del blocco. Il C supporta parecchi tipi fondamentali di oggetti:

Gli oggetti dichiarati come caratteri **(char)** sono abbastanza capaci da contenere un qualsiasi elemento del set di caratteri implementato, e se viene assegnato un carattere dal set in una variabile di questo tipo, il suo valore è equivalente al codice in numero intero di quel carattere. Si possono assegnare altre quantità alle variabili carattere, ma l'implementazione dipende dalla macchina.

È disponibile un massimo di tre dimensioni di interi, dichiarate **short int**, **int** e **long int**. Gli interi più lunghi occupano non meno della memoria di quelli più corti, ma l'implementazione può fare in modo che gli interi short o gli interi long od entrambi, siano equivalenti agli interi normali.

Gli interi "normali" hanno la naturale dimensione suggerita dalla architettura della macchina ospite; le altre dimensioni vengono fornite per soddisfare particolari esigenze.

Gli interi senza segno, dichiarati **unsigned**, seguono le regole, del modulo aritmetico 2^n in cui n è il numero di bit nella rappresentazione (nel PDP/11, non sono supportate le quantità long senza segno).

I numeri in virgola mobile in singola precisione **(float)** e quelli in doppia precisione **(double)** possono essere sinonimi in alcune implementazioni.

Poiché gli oggetti dei tipi precedenti possono essere utilmente interpretati come numeri, saranno indicati come tipi *aritmetici*. I tipi **char** e **int** di qualsiasi dimensione saranno collettivamente chiamati tipi *integral*. I **float** e **double** saranno collettivamente chiamati tipi **floating**.

Accanto ai fondamentali tipi aritmetici c'è una classe teoricamente infinita di tipi derivati ricavati dai tipi fondamentali nei seguenti modi:

array di oggetti della maggior parte dei tipi;
funzioni che ritornano un oggetto di un certo tipo;
puntatori ad oggetti di un certo tipo;
strutture contenenti una sequenza di oggetti di vari tipi;
unioni capaci di contenere qualsiasi oggetto tra i diversi oggetti di vario tipo.

In generale questi metodi di costruzione degli oggetti possono essere applicati ricorsivamente.

5. Oggetti e lvalues

Un *oggetto* è una regione di memoria manipolabile; un *lvalue* è un'espressione che si riferisce ad un oggetto. Un identificatore costituisce un banale esempio di un'espressione lvalue. Ci sono operatori che producono lvalues: per esempio, se **E** è un'espressione di tipo puntatore, allora ***E** è un'espressione lvalue che si riferisce all'oggetto a cui punta **E**. Il nome "lvalue" deriva dall'espressione di assegnamento **E1** = **E2** in cui l'operando sinistro **E1** dev'essere un'espressione lvalue. La seguente trattazione di ciascun operatore indica quando si prevede di avere operandi lvalues e in quali casi si produce un lvalue.

6. Conversioni

Un certo numero di operatori, a seconda dei propri operando, può causare la conversione del valore di un operando da un tipo all'altro. Questo paragrafo spiega il risultato che presumibilmente si ottiene da queste conversioni. Il paragrafo 6.6 riassume le conversioni eseguite dai più comuni operatori; sarà completata dalla trattazione di ogni singolo operatore.

6.1 Caratteri ed interi

Un carattere o un intero short può essere usato quando si può usare un intero. In tutti i casi il valore viene convertito in un intero. La conversione di un intero più corto in uno più lungo comporta sempre l'estensione del segno; gli interi sono quantità con segno. Il fatto che per i caratteri ci sia o non ci sia un'estensione del segno, dipende dalla macchina, ma è garantito che un elemento del set di caratteri standard non è mai negativo. Per quanto riguarda le macchine trattate in questo manuale, solo il PDP-11 estende il segno. Sul PDP/11, l'intervallo delle variabili carattere va da −128 a 127; i caratteri dell'alfabeto ASCII sono tutti positivi. Una costante carattere specificata con una notazione ottale subisce l'estensione del segno e può risultare negativa; per esempio, '**\377**' ha valore −1.
Quando un intero più ampio viene convertito in uno meno ampio o in un **char**, viene troncato sulla sinistra; i bit eccedenti vengono semplicemente scartati.

6.2 Float e double

In C tutta l'aritmetica in floating point viene eseguita in doppia precisione; ogni volta che un **float** appare in un'espressione, viene allungato a **double** inserendo degli zeri nella sua parte decimale. Quando si deve convertire un **double** in un **float**, per esempio tramite un assegnamento, il **double** viene arrotondato prima di essere troncato alla lunghezza del **float**.

6.3 Floating e integral

La conversione da valori floating in tipi integral tende ad essere piuttosto dipendente dalla macchina, specialmente la direzione del troncamento dei numeri negativi varia da macchina a macchina. Se il valore non può essere contenuto nello spazio previsto, il risultato è indefinito.

Le conversioni da valori integral in tipi floating sono ben supportate. Se la destinazione non dispone di sufficienti bit, la precisione viene in parte perduta.

6.4 Puntatori e interi

Un intero o un intero long possono essere sommati o sottratti da un puntatore, in questo caso il primo viene convertito come specificato nella trattazione dell'operatore di addizione.
Due puntatori ad oggetti dello stesso tipo possono essere sottratti; in questo caso il risultato viene convertito in un intero come specificato nella trattazione degli operatori di sottrazione.

6.5 Unsigned

Quando vengono combinati un intero unsigned ed un intero normale, l'intero normale viene convertito in unsigned ed il risultato è unsigned. Il valore è il minimo intero unsigned congruente all'intero con segno (modulo $2^{\text{lunghezza-parola}}$). In una rappresentazione in complemento a due, questa conversione è teorica e in effetti non c'è uno scambio nella rappresentazione a bit.
Quando un intero unsigned viene convertito in long, il valore del risultato è numericamente lo stesso dell'intero unsigned. Quindi la conversione viene ottenuta aggiungendo degli zeri a sinistra.

6.6 Conversioni aritmetiche

Un grande numero di operatori causano conversioni ed ottengono i tipi dei risultati in modo simile. Questo comportamento sarà chiamato "usuali conversioni aritmetiche".

Innanzitutto, ogni operando di tipo **char** o **short** viene convertito in **int**, mentre ogni operando di tipo **float** viene convertito in **double**.
Se poi uno dei due operandi è **double**, l'altro viene convertito in **double** e questo è il tipo del risultato.
Altrimenti, se uno dei due operandi è **long**, l'altro viene convertito in **long** e questo è il tipo del risultato.
Altrimenti, se uno dei due operandi è **unsigned**, l'altro viene convertito in **unsigned** e questo è il tipo del risultato.
Altrimenti, entrambi gli operandi devono essere **int**, e questo è il tipo del risultato.

7. Espressioni

L'ordine di precedenza degli operatori nelle espressioni è uguale all'ordine dei principali sottoparagrafi di questo paragrafo, dove il primo è il più importante. Per esempio, le espressioni indicate come operandi di + (paragrafo 7.4) sono le espressioni definite dal paragrafo 7.1 al paragrafo 7.3. All'interno di ogni sottoparagrafo, gli operatori hanno la stessa precedenza. In ogni sottoparagrafo si specifica se gli operatori trattati sono associati a sinistra o a destra. La precedenza e l'associatività di tutti gli operatori di espressione è riassunta nella grammatica del paragrafo 18.
In altri casi l'ordine di valutazione delle espressioni è indefinito. In particolare il compilatore si ritiene libero di calcolare le sottoespressioni nell'ordine che ritiene più efficiente, anche se le sottoespressioni comportano effetti collaterali. Non viene specificato l'ordine in cui si verificano gli effetti collaterali. Le espressioni che contengono un'operatore commutativo o associativo (*****, **+**, **&**, **|**, **^**) possono essere arbitrariamente riarrangiate,

anche in presenza di parentesi; per forzare un particolare ordine di valutazione bisogna usare un'esplicita variabile temporanea.
Nella valutazione di un'espressione il trattamento dell'overflow ed il controllo sulla divisione dipende dalla macchina. Tutte le implementazioni esistenti del C ignorano gli overflow di interi; la trattazione delle divisioni per zero così come tutte le eccezioni del floating point, che sono generalmente gestite attraverso funzioni di libreria, variano da macchina a macchina.

7.1 Espressioni primarie

Le espressioni primarie che chiamano in causa ., −>, indici e chiamate di funzioni, si raggruppano da sinistra a destra.

espressione-primaria:
- *identificatore*
- *costante*
- *stringa*
- (*espressione*)
- *espressione-primaria* [*espressione*]
- *espressione-primaria* (*elenco-di-espressioni*$_{opz}$)
- *lvalue-primario . identificatore*
- *espressione-primaria −> identificatore*

elenco-di-espressioni:
- *espressione*
- *elenco-di-espressioni, espressione*

Un identificatore è un'espressione primaria, sempre che sia stato dichiarato in maniera appropriata come discusso sotto. Il suo tipo viene specificato dalla propria dichiarazione. Se il tipo dell'identificatore è "array di ...", allora il valore dell'identificatore-espressione è un puntatore al primo oggetto dell'array ed il tipo dell'espressione è "puntatore a ...". Inoltre, un identificatore di tipo array non è un'espressione lvalue. Allo stesso modo, un identificatore che è dichiarato "funzione ritornante ...", quando viene usato, sempre che non si usi nella posizione del nome della funzione in una chiamata, viene convertito in "puntatore a funzione ritornante ...".
Una costante è un'espressione primaria. Il suo tipo può essere **int**, **long** o **double** a seconda della sua forma. Le costanti carattere hanno tipo **int**; le costanti floating sono **double**.

Una stringa è un'espressione primaria. Il suo tipo è originariamente "array di **char**"; ma poiché segue le stesse regole date prima per gli identificatori, viene modificato in "puntatore a **char**" e il risultato è un puntatore al primo carattere della stringa. (Esiste un'eccezione in certe inizializzazioni; vedi paragrafo 8.6).
Un'espressione tra parentesi è un'espressione primaria il cui tipo e valore sono identici a quelli dell'espressione senza parentesi. La presenza di parentesi non influisce sul fatto che l'espressione sia o no un lvalue.
Un'espressione primaria seguita da un'espressione tra parentesi quadre è un'espressione primaria. Il significato intuitivo è quello di un indice. Generalmente, l'espressione primaria ha tipo "puntatore a ...", l'espressione di indice è **int**, ed il tipo del risultato è "...". L'espressione **E1[E2]** è identica (per definizione) a ∗**((E1)**+**(E2))**. Tutte le indicazioni

necessarie per capire questa notazione sono contenute in questo paragrafo insieme alla discussione dei paragrafi 7.1, 7.2 e 7.4 rispettivamente sugli identificatori, * e +. Il paragrafo 14.3 ne riassume le implicazioni.
Una chiamata di funzione è un'espressione primaria seguita da parentesi contenenti un elenco (possibilmente vuoto) di espressioni separate da virgola che costituiscono gli argomenti passati alla funzione. L'espressione primaria dev'essere di tipo "funzione ritornante ...", ed il risultato della chiamata di funzione è di tipo "...". Come indicato sotto, un identificatore non ancora conosciuto seguito immediatamente da una parentesi aperta rappresenta una funzione ritornante un intero; perciò nella maggior parte dei casi non è necessario dichiarare le funzioni con valori interi.
Ogni argomento di tipo **float** viene convertito in **double** prima della chiamata; ogni argomento di tipo **char** o **short** viene convertito in **int**; e come al solito i nomi di array vengono convertiti in puntatore. Non vengono eseguite altre conversioni automatiche; in particolare, il compilatore non confronta i tipi degli argomenti presenti con quelli degli argomenti formali. Se si necessita una conversione, bisogna usare un cast; vedi i paragrafi 7.2, 8.7.
Nella preparazione di una chiamata a funzione, viene effettuata una copia di ognuno dei parametri presenti; perciò, in C ogni argomento viene passato strettamente per valore. Una funzione può cambiare i valori dei propri parametri formali, ma questi cambiamenti non incidono sui valori dei parametri reali. D'altro lato, è possibile passare un puntatore per fare in modo che la funzione possa cambiare il valore di un oggetto a cui punta il puntatore. Il nome di un array è un'espressione puntatore. L'ordine di valutazione degli argomenti non è definito dal linguaggio; i vari compilatori sono diversi. Sono permesse chiamate ricorsive a qualsiasi funzione.
Un'espressione primaria seguita da un punto seguito da un identificatore è un'espressione. La prima espressione dev'essere un lvalue che si riferisce ad una struttura o unione, e l'identificatore deve avere il nome di un membro della struttura o unione. Il risultato è un lvalue che si riferisce al membro nominato della struttura o unione.
Un'espressione primaria seguita da una freccia (costruita da un − e un >) seguita da un identificatore è un'espressione. La prima espressione dev'essere un puntatore a una struttura o unione e l'identificatore deve avere il nome di un membro della struttura o unione. Il risultato è un lvalue che si riferisce al membro nominato della struttura o unione a cui punta l'espressione puntatore.
Perciò l'espressione **E1 −>MOS** è identica a **(*E1).MOS.** Le strutture e le unioni vengono trattate nel paragrafo 8.5. Le regole che abbiamo dato per l'uso delle strutture e delle unioni non sono tassativamente imposte per permettere una certa flessibilità nei meccanismi di scrittura. Vedi il paragrafo 14.1.

7.2 Operatori unari

Le espressioni contenenti operatori unari si raggruppano da destra a sinistra.

espressione-unaria:
- ∗ *espressione*
- **&** *lvalue*
- − *espressione*
- **!** *espressione*
- ˜*espressione*
- ++ *lvalue*
- −− *lvalue*
- *lvalue* ++
- *lvalue* −−
- (*nome-di-tipo*) *espressione*
- **sizeof** *espressione*
- **sizeof** (*nome di tipo*)

L'operatore unario ∗ significa *indirezione*: l'espressione dev'essere un puntatore ed il risultato è un lvalue che si riferisce all'oggetto a cui punta l'espressione. Se il tipo dell'espressione è "puntatore a ...", il tipo del risultato è "...".
Il risultato dell'operatore unario **&** è un puntatore all'oggetto indicato da lvalue. Se il tipo di lvalue è "...", il tipo del risultato è "puntatore a ...".
Il risultato dell'operatore unario − è il negativo del suo stesso operando. Vengono eseguite le usuali conversioni aritmetiche. Il negativo di una quantità unsigned viene calcolato sottraendo il suo valore da 2^n, in cui *n* è il numero di bit in un **int**. Non esiste un operatore unario +.
Il risultato dell'operatore logico di negazione **!** è 1 se il valore dei suoi operandi è 0, mentre è 0 se il valore dei suoi operandi è diverso da zero. Il tipo del risultato è **int**. È applicabile a qualsiasi tipo aritmetico o ai puntatori.
L'operatore˜ produce il complemento a uno dei suoi operandi. Vengono eseguite le usuali conversioni aritmetiche. Il tipo dell'operando dev'essere integral.
L'oggetto a cui si riferisce l'operando lvalue dal prefisso ++ è incrementato. Il valore è il nuovo valore dell'operando, ma non è un lvalue. L'espressione ++**x** è equivalente a **x**+=**1**. Si veda la trattazione degli operatori di addizione (paragrafo 7.4) e di assegnamento (paragrafo 7.14) per avere informazioni sulle conversioni.
L'operando lvalue di prefisso −− viene decrementato analogamente all'operatore prefisso ++.
Quando il postfisso ++ viene applicato ad un lvalue, il risultato è il valore dell'oggetto indicato dallo lvalue. Una volta usato il risultato, l'oggetto viene incrementato nello stesso modo dell'operatore ++ prefisso. Il tipo del risultato è lo stesso tipo dell'espressione lvalue.
Quando viene applicato il postfisso −− ad un lvalue, il risultato è il valore dell'oggetto riferito dall'lvalue. Dopo che il risultato è usato, l'oggetto viene decrementato nello stesso modo dell'operatore −− prefisso. Il tipo del risultato è lo stesso del tipo dell'espressione lvalue.
Un'espressione preceduta da un nome tra parentesi di un tipo di dato causa la conversione del valore dell'espressione nel tipo nominato. Questa costruzione è chiamata un *cast*. I nomi dei tipi sono descritti nel paragrafo 8.7.
L'operatore **sizeof** riporta la dimensione, in byte, del suo operando. (Un byte è definito dal linguaggio solamente in termini di valore di **sizeof**. Comunque, in tutte le implementazioni esistenti un byte è lo spazio richiesto per un **char**. Se applicato ad un array, il risultato

è il numero totale di byte nell'array. La dimensione è determinata dalle dichiarazioni degli oggetti nell'espressione. Semanticamente questa espressione è una costante intera e può essere usata ovunque sia richiesta una costante. Il suo uso più comune è nelle comunicazioni con routine tipo allocatori di memoria e sistemi di I/O.
L'operatore **sizeof** può anche essere applicato a nomi di tipi tra parentesi. In questo caso rende la dimensione in byte di un oggetto del tipo indicato. La costruzione **sizeof** (*tipo*) rappresenta un'unità, quindi l'espressione **sizeof** (*tipo*)**–2** è uguale a (**sizeof** (*tipo*))**–2**

7.3 Operatori moltiplicativi

Gli operatori moltiplicativi ∗, / e **%** si raggruppano da sinistra a destra. Vengono eseguite le usuali conversioni aritmetiche.

espressione-moltiplicativa
espressione ∗ *espressione*
espressione / *espressione*
espressione **%** *espressione*

L'operatore binario ∗ indica moltiplicazione. L'operatore ∗ è associativo ed il compilatore è libero di riarrangiare espressioni con più moltiplicazioni allo stesso livello.
L'operatore binario / indica divisione. Quando vengono divisi interi positivi il troncamento è in direzione dello zero, ma se qualcuno degli operandi è negativo la forma del troncamento dipende dalla macchina. In tutte le macchine prese in considerazione in questo manuale, il resto ha lo stesso segno del dividendo. È sempre vero che **(a/b)∗b + a%b** è uguale ad **a** (se **b** non è zero).
L'operatore binario **%** produce il resto della divisione della prima espressione per la seconda. Vengono eseguite le usuali conversioni aritmetiche. Gli operandi non devono essere **float**.

7.4 Operatori additivi

Gli operatori additivi + e **–** si raggruppano da sinistra a destra. Vengono eseguite le usuali conversioni aritmetiche. Per ogni operatore ci sono delle altre possibilità.

espressione-additiva:
espressione + *espressione*
espressione **–** *espressione*

Il risultato dell'operatore + è la somma degli operandi. Si può sommare un puntatore ad un oggetto in un array con un valore di ogni tipo integrale. Quest'ultimo viene in ogni caso convertito in un indirizzo costruito moltiplicandolo per la lunghezza dell'oggetto a cui punta il puntatore. Il risultato è un puntatore dello stesso tipo del puntatore iniziale, che punta ad un altro oggetto nello stesso array, opportunamente determinato dall'oggetto originale. Perciò se **P** è un puntatore ad un oggetto in un array, l'espressione **P+1** è un puntatore al successivo oggetto dell'array.
Non sono disponibili altre combinazioni di tipi per i puntatori.
L'operatore + è associativo e le espressioni con più addizioni allo stesso livello possono essere riarrangiate dal compilatore.
Il risultato dell'operatore **–** è la differenza degli operandi. Vengono eseguite le usuali conversioni aritmetiche. Inoltre, si può sottrarre da un puntatore un valore di qualsiasi tipo integral, vengono applicate le stesse conversioni delle addizioni.

Se vengono sottratti due puntatori ad oggetti dello stesso tipo, il risultato viene convertito (dividendo per la lunghezza dell'oggetto) in un **int** che rappresenta il numero di oggetti che separano gli oggetti puntati. A meno che i puntatori puntino ad oggetti dello stesso array, generalmente questa conversione darà risultati inaspettati poiché i puntatori, anche ad oggetti dello stesso tipo, non differiscono necessariamente di un multiplo della lunghezza dell'oggetto.

7.5 Operatori di shift

Gli operatori di shift << e >> si raggruppano da sinistra a destra. Entrambi ottengono le usuali conversioni aritmetiche sui propri operandi, ognuno dei quali dev'essere integral. Successivamente l'operando destro viene convertito in **int**, il tipo del risultato è quello dell'operando sinistro. Se l'operando destro è negativo, o maggiore o uguale alla lunghezza dell'oggetto in bit, il risultato è indefinito.

espressione-di-shift:
espressione << espressione
espressione >> espressione

Il valore di **E1<<E2** è **E1** (interpretato come raggruppamento di bit) con uno shift a sinistra di **E2** bit; i bit vacanti sono riempiti con 0. Il valore di **E1>>E2** è **E1** con uno shift a destra di **E2** posizioni di bit. Lo shift a destra è certamente logico (riempito con 0) se **E1** è **unsigned**; altrimenti può essere (ed è, sul PDP/11) aritmetico (riempito con una copia del bit di segno).

7.6 Operatori relazionali

Gli operatori relazionali si raggruppano da sinistra a destra, ma ciò non è molto utile; **a<b<c** non significa ciò che sembra.

espressione-relazionale:
espressione < espressione
espressione > espressione
espressione <= espressione
espressione >= espressione

Gli operatori < (minore di), > (maggiore di), <= (minore o uguale a) e >= (maggiore o uguale a) danno tutti 0 se la relazione specificata è falsa ed 1 se è vera. Il tipo del risultato è **int**. Vengono eseguite le usuali conversioni aritmetiche. Due puntatori possono essere confrontati; il risultato dipende dalle relative locazioni nello spazio degli indirizzi degli oggetti puntati. Il confronto di puntatori è portabile solamente quando i puntatori puntano ad oggetti dello stesso array.

7.7 Operatori di eguaglianza

espressione-di-eguaglianza:
espressione == espressione
espressione != espressione

Gli operatori == (uguale a) e != (diverso da) sono esattamente uguali agli operatori relazionali a parte la loro minore precedenza. (Quindi **a<b == c<d** è 1 ogni qual volta che **a<b** e **c<d** hanno lo stesso valore di verità).

Un puntatore può essere confrontato con un intero ma il risultato è dipendente dalla macchina a meno che l'intero non sia la costante 0. Sicuramente un puntatore a cui è stato assegnato 0 non punterà a nessun oggetto, e sembrerà uguale a zero; nell'uso convenzionale un puntatore di questo tipo è considerato nullo.

7.8 Operatore AND sui bit

espressione-and
espressione **&** *espressione*

L'operatore **&** è associativo e le espressioni che hanno a che fare con **&** possono essere riarrangiate. Vengono eseguite le usuali conversioni aritmetiche; Il risultato è la funzione **AND** sui bit degli operandi. L'operatore si applica solamente agli operandi integral.

7.9 Operatore OR esclusivo sui bit

espressione-or-esclusivo:
epressione ^ *espressione*

L'operatore^ è associativo e le espressioni che hanno a che fare con^ possono essere riarrangiate. Vengono eseguite le usuali conversioni aritmetiche; Il risultato è la funzione di **OR** esclusivo sui bit degli operandi. L'operatore si applica solamente agli operandi integral.

7.10 Operatore OR inclusivo sui bit

espressione-or-inclusivo:
espressione | *espressione*

L'operatore | è associativo e le espressioni che hanno a che fare con | possono essere riarrangiate. Vengono eseguite le usuali conversioni aritmetiche; Il risultato è la funzione di **OR** inclusivo sui bit degli operandi. L'operatore si applica solamente agli operandi integrali.

7.11 Operatore AND logico

espressione-and-logico:
espressione **&&** *espressione*

L'operatore **&&** si raggruppa da sinistra a destra. Ritorna 1 se entrambi gli operandi sono diversi da zero, 0 negli altri casi. Diversamente da **&**, **&&** garantisce la valutazione da sinistra a destra; inoltre il secondo operando non viene valutato se il primo operando è 0.
Non è necessario che gli operandi abbiano lo stesso tipo, ma ciascuno deve avere uno dei tipi fondamentali o essere puntatori. Il risultato è sempre **int**.

7.12 Operatore OR logico

espressione-or-logico:
espressione | | *espressione*

L'operatore | | si raggruppa da sinistra a destra. Ritorna 1 se uno dei due operandi è diverso da zero, in caso contrario ritorna 0. Diversamente da |, | | garantisce la valutazione da sinistra a destra; inoltre il secondo operando non viene valutato se il primo operando è diverso da zero.
Non è necessario che gli operandi abbiano lo stesso tipo, ma entrambi devono avere uno dei tipi fondamentali o essere puntatori. Il risultato è sempre **int**.

7.13 Operatore condizionale

espressione-condizionale:
espressione **?** *espressione* **:** *espressione*

Le espressioni condizionali si raggruppano da destra a sinistra. Viene valutata la prima espressione e, se questa è diversa da zero, il risultato è il valore della seconda espressione, altrimenti è quello della terza espressione. Se è possibile, vengono eseguite le usuali conversioni aritmetiche per portare la seconda e la terza espressione in un tipo comune, se invece entrambi sono puntatori allo stesso tipo, il risultato ha il tipo comune, altrimenti, una dev'essere un puntatore e l'altra la costante 0, e il risultato ha il tipo del puntatore. Tra la seconda e la terza espressione se ne valuta una sola.

7.14 Operatori di assegnamento

Esiste un certo numero di operatori di assegnamento, ognuno dei quali si raggruppa da destra a sinistra. Richiedono tutti un lvalue come proprio operando sinistro, e un'espressione di assegnamento ha lo stesso tipo di quello del suo operando sinistro. Il valore è il valore memorizzato nell'operando sinistro dopo che è stato effettuato l'assegnamento. Le due parti di un operatore di assegnamento composto sono simboli separati.

espressione-di-assegnamento
lvalue = *espressione*
lvalue += *espressione*
lvalue –= *espressione*
lvalue *= *espressione*
lvalue /= *espressione*
lvalue **%**= *espressione*
lvalue >>= *espressione*
lvalue <<= *espressione*
lvalue **&**= *espressione*
lvalue ^= *espressione*
lvalue **!**= *espressione*

Nel semplice assegnamento con =, il valore dell'espressione sostituisce quello dell'og-

getto indicato dallo lvalue. Se entrambi gli operandi hanno tipo aritmetico, l'operando viene convertito nel tipo dell'elemento sinistro dell'assegnamento.
Il comportamento di un'espressione della forma **E1** *op*= **E2** si può dedurre considerandolo come equivalente a **E1** = **E1** *op* (**E2**); comunque, **E1** viene valutata solo una volta. In +=e −=, l'operando sinistro può essere un puntatore, e in questo caso l'operando destro (integrale) viene convertito come spiegato nel paragrafo 7.4; tutti gli operandi destri e tutti quelli sinistri che non sono puntatori devono avere tipi aritmetici.
Attualmente i compilatori permettono che un puntatore sia assegnato ad un intero, un intero ad un puntatore, ed un puntatore a un puntatore di altro tipo. L'assegnamento è una operazione di pura copia, senza conversioni. Questo uso non è portabile, e può produrre puntatori che quando vengono usati producono diversità di indirizzamento. In ogni caso, è certo che l'assegnamento della costante 0 ad un puntatore produrrà un puntatore nullo distinguibile da un puntatore ad un oggetto di qualsiasi tipo.

7.15 Operatore virgola

operatore-virgola:
espresione , *espressione*

Una coppia di espressioni separate da virgola è valutata da sinistra a destra e il valore dell'espressione di sinistra viene scartato. Il tipo ed il valore del risultato sono il tipo ed il valore dell'operando destro. Questo operatore raggruppa da sinistra a destra. In contesti in cui la virgola ha un significato particolare, per esempio in una lista di argomenti a funzioni (paragrafo 7.1) e negli elenchi di inizializzazioni (paragrafo 8.6), l'operatore virgola così come è stato descritto in questo paragrafo, può solamente apparire all'interno di parentesi; per esempio,

```
f(a, (t=3, t+2), c)
```

ha tre argomenti, il secondo dei quali ha valore 5.

8. Dichiarazioni

Le dichiarazioni vengono usate per specificare l'interpretazione che il C darà ad ogni identificatore; esse non devono necessariamente riservare memoria associata all'identificatore. Le dichiarazioni hanno la seguente forma:

dichiarazione:
specifica-dich elenco-di-dichiaratori$_{opz}$;

I dichiaratori nell'elenco contengono gli identificatori che stanno per essere dichiarati. Le specifiche-dich consistono in una sequenza di specificatori di tipo e di classe di memoria.

specifica-dich:
specifica-di-tipo specifica-dich$_{opz}$
specifica-clas-men specifica-dich$_{opz}$

Questo elenco deve essere coerente al suo interno nel modo descritto più avanti.

8.1 Specifiche di classe di memoria

Le specifiche-clas-mem sono:

specifica-clas-mem:

```
auto
static
extern
register
typedef
```

La specifica **typedef** non riserva alcuna memoria e viene chiamata una "specifica di classe di memoria" solamente per convenienza sintattica; viene descritta nel paragrafo 8.8.
Il significato delle varie classi di memoria è stato chiarito nel paragrafo 4.
Le dichiarazioni **auto**, **static** e **register** hanno anche la funzione di definizione in quanto permettono l'allocazione di un'appropriata quantità di memoria. Nel caso **extern**, dev'esserci una definizione esterna degli identificatori (paragrafo 10) al di fuori della funzione in cui sono dichiarati.
Una dichiarazione **register** è da considerarsi come una dichiarazione **auto**, in cui si segnala al compilatore che le variabili dichiarate saranno usate ampiamente. Di queste dichiarazioni sono valide soltanto alcune fra le prime. Inoltre entreranno nei registri solo le variabili di certi tipi; sul PDP/11, questi tipi sono **int**, **char** o puntatore. Sulle variabili register esiste un'altra restrizione: non si può applicare ad esse l'operatore di indirizzo **&**.
Se le dichiarazioni dei registri vengono usate propriamente, si otterranno programmi più piccoli e più veloci, ma gli sviluppi futuri nella generazione del codice potrebbero renderle superflue.
In una dichiarazione si può dare al massimo una specifica di classe di memoria. Se da una dichiarazione manca la specifica di classe di memoria, all'interno di una funzione si considera auto, e **extern** all'esterno. Eccezioni: le funzioni non sono mai automatiche.

8.2 Specifiche di tipo

Le specifiche-di-tipo sono:

specifiche-di-tipo:

```
char
short
int
long
unsigned
float
double
```

specifica-di-struttura-o-unione
nome-di-typedef

Le parole **long**, **short** e **unsigned** possono essere intese come aggettivi; sono accettabili le seguenti combinazioni:

```
short int
long int
unsigned int
long float
```

Il significato dell'ultimo è lo stesso di **double**. Altrimenti, in una dichiarazione si può dare al massimo una specifica di tipo. Se quest'ultima manca, viene considerato **int**.
Le specifiche per le strutture e le unioni sono descritte nel paragrafo 8.5. Le dichiarazioni con nomi typedef sono trattate nel paragrafo 8.8.

8.3 Dichiaratori

L'elenco dei dichiaratori che compare in una dichiarazione è una sequenza di dichiaratori separati da virgola, ognuno dei quali può avere un inizializzatore.

elenco-dichiaratori:
    dichiaratore-iniz
    dichiaratore-iniz, elenco-dichiaratori
dichiaratore-iniz:
    dichiaratore inizializzatore$_{opz}$

Gli inizializzatori sono trattati nel paragrafo 8.6. Le specificazioni nella dichiarazione indicano il tipo e la classe di memoria degli oggetti a cui si riferiscono i dichiaratori. I dichiaratori hanno la seguente sintassi:

dichiaratore:
    identificatore
    (*dichiaratore* **)**
    ∗ *dichiaratore*
    dichiaratore **()**
    dichiaratore **[** *espressione-costante*$_{opz}$ **]**

Il raggruppamento è lo stesso delle espressioni.

8.4 Significato dei dichiaratori

Ogni dichiaratore è considerato come un'asserzione che produce un oggetto del tipo e della classe di memoria indicati quando in un'espressione appare un costrutto della stessa forma del dichiaratore. Ogni dichiaratore contiene esattamente un identificatore; è questo identificatore che viene dichiarato.
Se un identificatore senza parentesi appare come un dichiaratore, allora ha il tipo indicato dalla specifica in testa alla dichiarazione.
Un dichiaratore tra parentesi è identico al dichiaratore senza parentesi, ma le parentesi possono alterare il legame tra dichiaratori complessi.
Analizziamo ora la dichiarazione.

```
T D1
```

in cui **T** è una specifica di tipo (come **int**, ecc.) e **D1** è un dichiaratore. Supponiamo che questa dichiarazione faccia in modo che l'identificatore abbia tipo "... **T**", in cui "..." è vuoto se **D1** è solamente un identificatore "pieno" (cioè, il tipo di **x** in "**int x**" è solamente **int**. Allora se **D1** ha la forma

```
*D
```

il tipo dell'identificatore contenuto è "... puntatore a **T**".
Se **D1** ha la forma

```
D()
```

allora l'identificatore contenuto è del tipo "... funzione ritornante **T**".

Se **D1** ha la forma

D[*espressione-costante*]

oppure

```
D[ ]
```

allora l'identificatore contenuto è del tipo "... array di **T**". Nel primo caso l'espressione costante è un'espressione il cui valore è determinabile in compilazione ed il cui tipo è **int**. (Le espressioni costanti sono definite più precisamente nel paragrafo 15). Quando compaiono più specifiche "array di" adiacenti, viene creato un array a più dimensioni; le espressioni costanti che specificano i confini dell'array possono essere tralasciate solamente nel primo membro della sequenza. L'omissione è utile quando l'array è esterno e la definizione vera e propria, che alloca memoria, viene data altrove. Inoltre la prima espressione costante può essere omessa quando il dichiaratore è seguito da un'inizializzazione. In questo caso l'ampiezza viene calcolata dal numero di elementi inizializzati.
Un array può essere costruito da un tipo basilare, da un puntatore, da una struttura od unione o da un'altro array (per generare un array a più dimensioni).
In effetti non sono permesse tutte le possibilità date dalla precedente sintassi. Le restrizioni sono le seguenti: le funzioni non possono ritornare array, strutture, unioni o funzioni, sebbene possano ritornare puntatori a tali oggetti; non sono possibili array di funzioni, sebbene ci possano essere array di puntatori a funzioni. Allo stesso modo una struttura o un'unione non può contenere una funzione, ma può contenere un puntatore a funzione.
Per esempio, la dichiarazione

```
int i, *ip, f(), *fip(), (*pfi)();
```

dichiara un intero **i**, un puntatore **ip** ad un intero, una funzione **f** che ritorna un intero, una funzione **fip** che ritorna un puntatore ad un intero, ed un puntatore **pfi** ad una funzione che ritorna un intero. È particolarmente utile confrontare gli ultimi due. Il legame di ***fip()** è ***(fip())**, così come la dichiarazione esprime e la costruzione stessa dell'espressione richiede la chiamata di una funzione **fip**; e usando poi un'indirezione per mezzo del risultato (puntatore) si ottiene un intero. Nel dichiaratore **(*fpi)()**, le

parentesi aggiunte sono necessarie, come lo sono nelle espressioni, per indicare che l'indirezione attraverso un puntatore ad una funzione produce una funzione, che viene successivamente chiamata; ritorna un intero.
Per fare un altro esempio,

```
float fa[17], *afp[17];
```

dichiara un array di numeri **float** ed un array di puntatori a numeri **float**. Alla fine,

```
static int x3d[3][5][7];
```

dichiara un array statico a tre dimensioni di interi, con ampiezza 3×5×7. Entrando nei dettagli, **x3d** è un array di tre elementi; ogni elemento è un array di 5 array; ognuno di questi ultimi array è un array di 7 interi. In un'espressione può certamente apparire ognuna delle espressioni **x3d, x3d[i], x3d[i][j], x3d[i][j][k].** Le prime tre hanno tipo "array", l'ultima ha tipo **int**.

8.5 Dichiarazione di strutture e unioni

Una struttura è un oggetto che consiste in una sequenza di membri con nome. Ogni membro può essere di qualsiasi tipo. Un'unione è un oggetto che ad un certo punto può contenere uno qualsiasi dei numerosi membri. Le specifiche di strutture e unioni hanno la stessa forma.

specifica-strut-o-unione:
    strut-o-unione { *elenco-dich-strut* }
    identif-strut-o-unione { *elenco-dich-strut* }
    identif-strut-o-unione

strut-o-unione:

    struct
    union

L'elenco-dich-strut è una sequenza di dichiarazioni riguardanti i membri della struttura o dell'unione.

elenco-dich-strut:
    dichiarazione-strut
    dichiarazione-strut elenco-dich-strut
dichiarazione-strut:
    specifica-tipo elenco-dichiaratore-strut;
elenco-dichiaratore-strut:
    dichiarazione-strut
    dichiaratore-strut, elenco-dichiaratore-strut

Generalmente, un dichiaratore-strut è solamente un dichiaratore che riguarda il membro di una struttura o di un'unione. Il membro di una struttura può anche consistere di un numero specificato di bit. Un tale membro è anche chiamato *campo*; la sua lunghezza

è indicata dopo il nome del campo ed un due punti.

dichiaratore-strut:
dichiaratore
dichiaratore **:** *espressione-costante*
: *espressione-costante*

All'interno di una struttura, gli oggetti dichiarati hanno indirizzi in ordine crescente in quanto le loro dichiarazioni vengono lette da destra a sinistra. Ogni membro di una struttura che non è un campo inizia ad un indirizzo appropriatamente calcolato dal proprio tipo; perciò in una struttura ci possono essere spazi senza nome. I membri dei campi vengono raggruppati all'interno degli interi della macchina; essi non si estendono oltre le parole. Un campo che non si adatta nello spazio rimanente di una parola viene inserito nella parola successiva. Nessun campo può essere più ampio di una parola. I campi vengono assegnati da destra a sinistra nel PDP/11, da sinistra a destra nelle altre macchine.
Un dichiaratore-strut senza dichiaratore, ma solamente con i due punti e una ampiezza, indica un campo senza nome utile a riempire una certa ampiezza adattandosi a disposizioni imposte dall'esterno. In un caso particolare, un campo senza nome con una larghezza 0 specifica l'allineamento del campo successivo al limite di una parola. Il "campo successivo" è presumibilmente un campo, non un membro normale di struttura, poiché in quest'ultimo caso l'allineamento sarebbe automatico.
Il linguaggio non limita i tipi degli oggetti che vengono dichiarati come campi, ma le implementazioni non sono idonee a supportare nient'altro che campi interi. Inoltre, anche i campi int possono essere considerati unsigned. Sul PDP/11, i campi non hanno segno ed hanno solo valori interi. In tutte le implementazioni, non esistono array di campi, e non si può applicare ad essi l'operatore di indirizzo **&**, per cui non esistono puntatori a campi.
Un'unione può essere intesa come una struttura in cui tutti i membri iniziano a 0 e la cui dimensione è sufficiente a contenere qualsiasi suo membro. In qualsiasi momento in un'unione può entrare un solo membro.
Una specifica di struttura o di unione della seconda forma, cioè una delle

struct *identificatore* { *elenco-dich-strut* }
union *identificatore* { *elenco-dich-strut* }

dichiara che l'identificatore è il *modello della struttura* (o modello dell'unione) della struttura specificata nell'elenco. Una successiva dichiarazione può usare la terza forma di specifica, una delle

struct *identificatore*
union *identificatore*

I modelli di struttura permettono di definire le strutture ricorsive; inoltre permettono di usare più volte un'unica lunga dichiarazione. Non è permesso dichiarare una struttura o un'unione che contengono un riferimento a se stesse, ma una struttura o un'unione possono contenere un puntatore a se stesse.

I nomi dei membri e dei modelli possono essere gli stessi delle variabili ordinarie. Comunque, i nomi dei modelli e dei membri devono essere distinti tra di loro.
Due strutture possono condividere un'iniziale sequenza di membri; cioè, lo stesso membro può apparire in due differenti strutture se ha lo stesso tipo in entrambe e se anche tutti i membri precedenti sono uguali in entrambe (in effetti il compilatore controlla solamente che un nome in due strutture differenti abbia lo stesso tipo e posizione in entrambe, ma se i membri precedenti hanno una diversa costruzione non è portabile).
Un semplice esempio di una dichiarazione di struttura è

```
struct alb_nodo {
    char parola[20];
    int conta;
    struct alb_nodo *sinistra;
    struct alb_nodo *destra;
};
```

che contiene un array di 20 caratteri, un intero e due puntatori a strutture simili. Dopo che è stata data questa dichiarazione, la dichiarazione

```
struct alb_nodo s, *sp;
```

dichiara che **s** è una struttura del tipo dato e **sp** è un puntatore ad una struttura dello stesso tipo. Con queste dichiarazioni, l'espressione

```
sp ->conta
```

si riferisce al campo **conta** della struttura a cui punta **sp**;

```
s.sinistra
```

si riferisce al puntatore del sottoalbero sinistro della sttuttura **s**; e

```
s.destra->parola[0]
```

si riferisce al primo carattere del membro **parola** del sottoalbero destro di **s**.

8.6 Inizializzazione

Un dichiaratore può specificare un valore iniziale per l'identificatore che sta dichiarando. L'inizializzatore è preceduto da =, e consiste in un'espressione o in un elenco di valori nidificati tra graffe.

inizializzatore:
    = *espressione*
    = { *elenco-inizializzatori* }
    = { *elenco-inizializzatori,* }
elenco-inizializzatori:
    espressione
    elenco-inizializzatori, elenco-inizializzatori
    { *elenco-inizializzatori* }

Tutte le espressioni in un'inizializzatore di una variabile statica o esterna devono essere espressioni costanti, che sono descritte nel paragrafo 15, o espressioni che si riducono ad un indirizzo di una variabile precedentemente dichiarata, che può essere variato attraverso un'espressione costante. Le variabili automatiche o registri possono essere inizializzate da arbitrarie espressioni che richiamano costanti, e da variabili e funzioni precedentemente dichiarate.
Le variabili statiche ed esterne che non vengono inizializzate, hanno sicuramente valore iniziale uguale a zero; le variabili automatiche e registri che non sono inizializzate, hanno sicuramente valore iniziale casuale.
Quando si applica un inizializzatore ad uno *scalare* (un puntatore o un oggetto di tipo aritmetico), abbiamo un'unica espressione, magari tra graffe. Il valore iniziale dell'oggetto viene preso dall'espressione; vengono eseguite le stesse conversioni dell'assegnamento.
Quando la variabile dichiarata è un'*aggregata* (una struttura o un array) allora l'inizializzatore consiste in un elenco di inizializzatori per i membri della aggregata, racchiusi tra graffe e separati da virgola; l'elenco è scritto in ordine crescente di indice o membri. Se l'aggregata contiene sottoaggregate, questa regola si applica ricorsivamente ai membri dell'aggregata. Se nell'elenco ci sono meno inizializzatori dei membri presenti nell'aggregata, allora l'aggregata viene riempita con degli zeri. Non è permesso inizializzare aggregate di unioni o automatiche.
Le graffe possono essere omesse come segue. Se l'inizializzatore comincia con una graffa aperta, allora la lista di inizializzatori separati da virgola che la segue, inizializza i membri dell'aggregata; è un errore se ci sono più inizializzatori che membri. Se, invece, l'inizializzatore non inizia con una graffa aperta, allora vengono presi dalla lista solo gli elementi necessari e sufficienti per i membri dell'aggregata; ogni membro rimanente viene usato per inizializzare il successivo membro dell'aggregata di cui fa parte l'aggregata corrente.
Un'ultima abbreviazione permette di inizializzare un array di **char** con una stringa. In questo caso i successivi caratteri della stringa inizializzano i membri dell'array.
Per esempio,

```
int x[] = { 1, 3, 5 };
```

dichiara e inizializza x come un array a una dimensione che ha tre membri, poiché non è stata specificata la dimensione e ci sono tre inizializzatori.

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

è un'inizializzazione completamente tra graffe: 1, 3 e 5 inizializzano la prima riga dell'array **y[0],** chiamati **y[0][0], y[0][1]** e **y[0][2]**. Allo stesso modo le due linee successive inizializzano **y[1]** e **y[2]**. L'inizializzatore si ferma anzitempo e perciò **y[3]** è inizializzato con 0. Lo stesso effetto sarebbe stato ottenuto da

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

L'inizializzatore per y inizia con una graffa aperta, ma quella per **y[0]** no, per cui si usano tre elementi dell'elenco. Allo stesso modo altri tre vengono successivamente presi per **y[1]** e **y[2]**. Inoltre,

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

inizializza la prima colonna di **y** (considerato come un array a due dimensioni) e lascia il resto a 0.
Alla fine,

```
char mesg[] = "Errore di sintassi alla linea %d\n";
```

mostra un array di caratteri i cui membri vengono inizializzati con una stringa.

8.7 Nomi di tipo

In due contesti (per specificare le conversioni di tipo esplicitamente per mezzo di un cast e come argomento di **sizeof**) è preferibile dare il nome di un tipo di dato. Questo è possibile usando un "nome di tipo", che essenzialmente è una dichiarazione di un oggetto di quel tipo che omette il nome dell'oggetto.

nome-di-tipo:
    specificatore-di-tipo dichiaratore-astratto

dichiaratore-astratto:
    vuoto
    (*dichiaratore-astratto*)
    ∗ *dichiaratore-astratto* ()
    dichiaratore-astratto [*espressione-costante*$_{opz}$]

Per evitare ambiguità, nella costruzione

(*dichiaratore-astratto*)

si richiede che il dichiaratore-astratto non sia vuoto. Con questa restrizione, è possibile identificare inequivocabilmente la locazione del dichiaratore-astratto in cui apparirà l'identificatore se la costruzione sarà un dichiaratore in una dichiarazione. Il tipo nominato è allora lo stesso del tipo dell'ipotetico identificatore. Per esempio,

```
int
int *
int *[3]
int (*)[3]
int *()
int (*)()
```

chiamano i tipi rispettivamente "intero", "puntatore ad intero", "array di 3 puntatori ad interi", "puntatore ad un array di tre interi", "funzione ritornante un puntatore a un intero" e "puntatore a funzione ritornante un intero".

8.8 Typedef

Dichiarazioni la cui "classe di memoria" è **typedef** non definiscono memoria, ma definiscono invece identificatori che possono essere successivamente usati come se fossero parole chiave che nominano i tipi fondamentali o derivati.

nome-typedef:
identificatore

Nell'ambito della visibilità di una dichiarazione che coinvolge **typedef**, ogni identificatore che appare come parte di qualsiasi dichiaratore in quel contesto diventa sintatticamente equivalente alla parola chiave del nome del tipo associato con l'identificatore nel modo descritto nel paragrafo 8.4. Per esempio, dopo

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im; } complex;
```

le costruzioni

```
MILES distanze;
extern KLICKSP metricp;
complex z, *zp;
```

sono tutte dichiarazioni valide; il tipo di **distanze** è **int**, quello di **metricp** è "puntatore ad **int**", e quello di **z** è la struttura specificata. **zp** è un puntatore ad una tale struttura. **typedef** non introduce nuovi tipi, solamente sinonimi dei tipi che potrebbero essere specificati in altro modo.
Perciò nell'esempio precedente si considera che **distanze** abbia esattamente lo stesso tipo di un altro oggetto int.

9. Istruzioni

Se non è altrimenti indicato, le istruzioni vengono eseguite in sequenza.

9.1 Istruzione di espressione

Molte istruzioni sono istruzioni espressioni, che hanno la forma

espressione;

Le istruzioni espressioni sono generalmente assegnamenti o chiamate di funzione.

9.2 Istruzioni composte o blocco

Per poter usare più istruzioni al posto di una, viene fornita l'istruzione composta (chiamata anche "blocco"):

istruzione-composta:
{ *elenco-dichiarazioni*$_{opz}$ *elenco-istruzioni*$_{opz}$ }
elenco-dichiarazioni:
dichiarazione
dichiarazione elenco-dichiarazioni
elenco-istruzioni:
istruzione
istruzione elenco-istruzioni

Se qualsiasi identificatore nell'elenco di dichiarazioni è già stato dichiarato, la dichiarazione più esterna viene abbandonata per la durata del blocco per riprendere poi la sua efficacia.
Qualsiasi inizializzazione di variabili **auto** o **register** viene effettuata ogni volta che si entra all'inizio del blocco. È anche possibile (ma non è una buona pratica) trasferirle all'interno di un blocco; in questo caso non si eseguono le inizializzazioni. Le inizializzazioni di variabili **static** vengono eseguite solo una volta quando il programma inizia l'esecuzione. All'interno di un blocco, le dichiarazioni **extern** non allocano memoria, per cui non è permessa inizializzazione.

9.3 Istruzione condizionale

Le due forme di un'istruzione condizionale sono

if (*espressione*) *istruzione*
if (*espressione*) *istruzione* **else** *istruzione*

In entrambi i casi si valuta l'espressione e, se questa è diversa da zero, si esegue la prima sottoistruzione. Nel secondo caso si esegue la seconda sottoistruzione se l'espressione è zero. Come al solito l'ambiguità di "else" viene risolta mettendo in relazione un **else** all'ultimo **if**-senza-**else** incontrato.

9.4 Istruzione while

L'istruzione **while** ha la forma

while (*espressione*) istruzione

La sottoistruzione viene eseguita in ripetizione fino a che il valore dell'espressione rimane diverso da zero. Il test viene effettuato prima di ogni esecuzione dell'istruzione.

9.5 Istruzione do

L'istruzione **do** ha la forma

do *istruzione* **while** (*espressione*) **;**

La sottoistruzione viene eseguita in ripetizione fino a che il valore dell'espressione diventa zero. Il test viene effettuato dopo ogni esecuzione dell'istruzione.

9.6 Istruzione for

L'istruzione **for** ha la forma

for (*espressione-1*$_{opz}$; *espressione-2*$_{opz}$; *espressione-3*$_{opz}$) *istruzione*

Questa istruzione è equivalente a

espressione-1 ;
while (*espressione-2*) {
istruzione
espressione-3 ;
}

Perciò la prima espressione specifica l'inizializzazione per il ciclo; la seconda specifica un test, eseguito prima di ogni operazione, in maniera che il ciclo viene terminato quando l'espressione diventa zero; la terza espressione specifica spesso un incremento che viene effettuato dopo ogni iterazione.
Una o tutte le espressioni possono essere tralasciate. Se manca espressione-2, il **while** implicato diventa equivalente a **while**(1); altre espressioni mancanti vengono semplicemente tralasciate dall'espansione sopra indicata.

9.7 Istruzione switch

L'istruzione **switch** fa in modo che il controllo venga trasferito ad una o più istruzioni a seconda del valore dell'espressione. Ha la forma

switch (*espressione*) *istruzione*

Vengono eseguite le usuali conversioni aritmetiche sull'espressione, ma il risultato dev'essere **int**. L'istruzione è generalmente composta. Ogni istruzione all'interno dell'istruzione può essere etichettata con uno o più prefissi come segue:

case *espressione-costante* **:**

in cui l'espressione costante dev'essere **int**. Due costanti nello stesso switch non possono avere lo stesso valore. Le espressioni costanti sono meglio definite nel paragrafo 15.
Può anche comparire una sola istruzione della forma

default:

Quando viene eseguita l'istruzione **switch**, viene valutata la sua espressione e confrontata con le costanti dei "case". Se una delle costanti è uguale al valore dell'espressione, il controllo viene passato all'istruzione che segue quel prefisso case. Se nessuna costante incontra l'espressione, e se c'è il prefisso **default**, il controllo viene passato

all'istruzione che lo segue. Se non c'è nessun caso valido e se non è presente **default**, allora non viene eseguita nessuna istruzione dello switch.
I prefissi **case** e **default** in sé non alterano il flusso di controllo, che continua libero attraverso i prefissi. Per uscire da una switch, vedi **break**, paragrafo 9.8.
Generalmente l'istruzione che fa parte di uno switch, è composta. Possono apparire dichiarazioni in testa a questa istruzione, ma le inizializzazioni di variabili automatiche o registri non hanno alcun effetto.

9.8 Istruzione break

L'istruzione

break ;

provoca l'interruzione dell'istruzione **while, do, for** o **switch** più interna racchiusa tra graffe; il controllo passa all'istruzione che segue l'istruzione interrotta.

9.9 Istruzione continue

L'istruzione

continue ;

fa in modo che il controllo passi nella parte di continuazione-ciclo della più interna istruzione **while, do** o **for** in cui è racchiusa; cioè alla fine del ciclo. Più precisamente, in ognuna delle istruzioni

```
while (...) {          do {                  for (...) {
    ...                    ...                   ...
contin: ;              contin: ;             contin: ;
}                      } while (...);        }
```

un **continue** è equivalente a **goto contin**. (Ciò che segue **contin**: è un'istruzione nulla, paragrafo 9.13).

9.10 Istruzione return

Una funzione ritorna al suo chiamante attraverso il significato dell'istruzione return, che può avere una delle forme

return ;
return *espressione*;

Nel primo caso il valore di ritorno è indefinito. Nel secondo caso, il valore dell'espressione viene ritornato al chiamante della funzione. Se richiesto, l'espressione viene convertita, come per un assegnamento, nel tipo della funzione in cui appare. Raggiungere la fine di una funzione è equivalente ad un return senza valori di ritorno.

9.11 Istruzione goto

Il controllo può essere trasferito incondizionatamente per mezzo dell'istruzione

goto *identificatore*;

L'identificatore dev'essere una label (paragrafo 9.12) situata nella funzione corrente.

9.12 Istruzione con label

Ogni istruzione può essere preceduta da una label della forma

identificatore:

che ha lo scopo di dichiarare l'identificatore come una label. L'unico uso di una label è quello di essere l'obiettivo di un **goto**. La visibilità di una label è la funzione corrente, escludendo ogni sottoblocco in cui è stato ridichiarato lo stesso identificatore. Vedi paragrafo 11.

9.13 Istruzione nulla

L'istruzione nulla ha la forma

;

Un'istruzione nulla è utile per forzare una label appena prima della } di un'istruzione composta o per fornire un contenuto nullo a un'istruzione ciclica tipo **while**.

10. Definizioni esterne

Un programma C consiste in una sequenza di definizioni esterne. Una definizione esterna dichiara che un identificatore ha classe di memorizzazione **extern** (per default) oppure **static** ed un tipo specificato. La specifica del tipo (paragrafo 8.2) può anche essere vuota, in tal caso il tipo è considerato **int**. Le definizioni esterne sono visibili fino alla fine del file in cui vengono dichiarate proprio come l'effetto delle dichiarazioni che continua fino alla fine di un blocco. La sintassi delle definizioni esterne è uguale a quella di tutte le altre dichiarazioni, eccetto che il codice delle funzioni può essere dato solo a questo livello.

10.1 Definizioni di funzioni esterne

Le definizioni di funzioni hanno la forma

definizione-di-funzione
    specifica-dich$_{opz}$ *dichiarazione-funzione corpo-funzione*

Le uniche specifiche di classe di memoria permesse durante le specifiche di dichiarazione sono **extern** o **static**; vedi il paragrafo 11.2 per le differenze tra le due. Un dichiaratore di funzione è simile ad un dichiaratore per una "funzione che ritorna ..." eccetto che elenca i parametri formali della funzione che si sta definendo.

dichiaratore-di-funzione
    dichiaratore (elenco-di-parametri$_{opz}$*)*

elenco-di-parametri
    identificatore
    identificatore, elenco-di-parametri

Il corpo della funzione ha la forma

corpo-di-funzione:
elenco-dichiarazioni istruzione-composta

Gli identificatori nell'elenco dei parametri, e soltanto quelli, possono essere dichiarati nell'elenco di dichiarazioni. Ogni identificatore di cui non viene dato il tipo, viene considerato **int**. **register** è l'unica classe di memoria che può essere specificata; se è specificata, il corrispondente parametro viene copiato, se possibile, in un registro esterno alla funzione.
Un semplice esempio di una completa definizione di funzione è

```
int max(a, b, c)
int a, b, c;
{
    int m;

    m = (a > b) ? a : b;
    return((m > c) ? m : c);
}
```

Qui **int** è la specifica-di-tipo; **max(a, b, c)** è il dichiaratore di funzione; **int a, b, c**; è l'elenco di dichiarazioni per i parametri formali; { ... } è il blocco che fornisce il codice per l'istruzione.
Il C converte tutti i parametri di tipo **float** in **double**, così i parametri formali dichiarati **float** sono uniformati a **double** con una modifica delle loro dichiarazioni. Inoltre, poiché un riferimento ad un array in qualsiasi contesto (in particolare come parametro reale) significa un puntatore al primo elemento dell'array, le dichiarazioni dei parametri formali dichiarati "array di ..." vengono modificate in "puntatore a ...". Infine, poiché strutture, unioni e funzioni non possono essere passate a una funzione, è inutile dichiarare un parametro formale come struttura, unione o funzione (naturalmente sono permessi puntatori a tali oggetti).

10.2 Definizioni di dati esterni

Una definizione di dati esterni ha la forma

definizione-dati:
dichiarazione

La classe di memoria di tale tipo di dati dev'essere **extern** (che è per default) o **static**, ma non **auto** o **register**.

11. Regole di visibilità

Non c'è bisogno che un programma C venga compilato tutto in una volta: si può mantenere il testo sorgente del programma in più files, e si possono caricare dalle librerie routines precompilate. La comunicazione tra le funzioni di un programma può avvenire sia tramite chiamate esplicite che tramite la manipolazione di dati esterni.
Comunque, ci sono due tipi di visibilità, da considerare: primo, quella che possiamo chiamare la *visibilità lessicale* di un identificatore, che è essenzialmente la porzione del

programma in cui può essere usato senza provocare diagnostiche del tipo "identificatore indefinito"; secondo, la visibilità associata agli identificatori esterni, che è caratterizzata dalla regola che i riferimenti allo stesso identificatore esterno sono riferimenti allo stesso oggetto.

11.1 Visibilità lessicale

La visibilità lessicale degli identificatori dichiarati nelle definizioni esterne inizia alla definizione e finisce alla fine del file sorgente in cui appaiono. La visibilità lessicale degli identificatori che sono parametri formali persiste attraverso tutta la funzione con cui sono associati. La visibilità lessicale degli identificatori dichiarati in testa ad un blocco continua fino alla fine del blocco. La visibilità lessicale delle label è l'intera funzione in cui appaiono.
Poiché tutti i riferimenti allo stesso identificatore esterno si riferiscono allo stesso oggetto (paragrafo 11.2), il compilatore controlla tutte le dichiarazioni dello stesso identificatore esterno per verificarne la compatibilità; in effetti la loro visibilità viene estesa all'intero file in cui appaiono.
Comunque, in tutti i casi, se un identificatore viene esplicitamente dichiarato in testa ad un blocco, compreso un blocco che costituisce una funzione, si interrompe qualsiasi dichiarazione dell'identificatore all'esterno del blocco fino alla fine del blocco.
Da ricordare inoltre (paragrafo 8.5) che gli identificatori associati a variabili ordinarie da una parte e quelli associati a membri di strutture e unioni e prefissi dall'altra formano due classi distinte che non generano conflitti. I membri ed i prefissi seguono le stesse regole di visibilità degli altri identificatori. I nomi **typedef** sono nella stessa classe degli identificatori ordinari. Essi possono essere ridichiarati in blocchi interni ma nelle dichiarazioni interne bisogna dare esplicitamente un tipo:

```
typedef float distance;
...
{
    auto int distance;
    ...
```

Nella seconda dichiarazione **int** dev'essere presente altrimenti verrebbe presa per una dichiarazione senza dichiaratore e di tipo **distance**.

11.2 Visibilità delle esterne

Se una funzione si riferisce ad un identificatore dichiarato come **extern**, allora da qualche parte attraverso i files o le librerie che costituiscono l'intero programma deve esserci una definizione esterna dell'identificatore. Tutte le funzioni in un dato programma che si riferiscono allo stesso identificatore esterno si riferiscono allo stesso oggetto, per cui bisogna aver cura che il tipo e la dimensione specificata nella definizione siano compatibili con quelli specificati da ogni funzione che si riferisce ai dati.
La presenza della parola chiave **extern** in una definizione esterna indica che la memoria per gli identificatori che si stanno dichiarando sarà allocata in un'altro file. In questo modo in un programma su più files una definizione di dati esterni senza lo specificatore **extern** deve comparire in un solo file. Ogni altro file che vuole dare una definizione esterna per l'identificatore, deve includere **extern** nella definizione. L'identificatore può essere inizializzato solo nella dichiarazione in cui viene allocata la memoria.

Gli identificatori dichiarati **static** in testa alle definizioni esterne non sono visibili negli altri files. Le funzioni possono essere dichiarate **static**.

12. Linee di controllo del compilatore

Il compilatore C contiene un preprocessore con possibilità di sostituzione di macro, di compilazione condizionale e inclusione di files nominati. Le linee che iniziano con # comunicano con questo preprocessore. Queste linee hanno una sintassi indipendente dal resto del linguaggio; possono apparire ovunque ed hanno effetti (indipendentemente dalla visibilità) che si protraggono fino alla fine del file contenente il programma sorgente.

12.1 Sostituzione di simboli

Una linea di controllo del compilatore della forma

#**define** *identificatore stringa-di-simboli*

(da notare: senza punto e virgola alla fine) fa in modo che il preprocessore sostituisca le successive istanze dell'identificatore con una data stringa di simboli. Una linea della forma

#**define** *identificat (identificat,...,identificat) stringa-di-simboli*

in cui non c'è spazio tra il primo identificatore e (, è una definizione di macro con argomenti. Le istanze successive del primo identificatore seguite da (, una sequenza di simboli delimitati dalla virgola, ed una) vengono sostituiti nella definizione da una stringa di simboli. Ogni occorrenza dell'identificatore menzionato nell'elenco dei parametri formali della definizione viene sostituito nella chiamata dalla corrispondente stringa di simboli; nella chiamata i veri argomenti sono stringhe di simboli separati da virgola, comunque le virgole all'interno delle stringhe tra apici o racchiuse tra parentesi non separano gli argomenti. Il numero di parametri formali e reali deve essere lo stesso.
Il testo all'interno di una costante stringa o carattere non è soggetto a sostituzione.
In entrambe le forme la stringa di sostituzione viene scandita nuovamente per cercare altri identificatori definiti. In entrambe le forme una definizione lunga può proseguire sulla linea successiva inserendo \ alla fine della riga da continuare.
Questa facilitazione è particolarmente apprezzabile nella definizione di "costanti esplicative", come in

```
#define TABAMPIO 100

int table[TABAMPIO];
```

Una riga di controllo della forma

#**undef** *identificatore*

fa in modo che il preprocessore dimentichi la definizione dell'identificatore.

12.2 Inclusione di file

Una linea di controllo del compilatore della forma

#include *"nome-file"*

causa la sostituzione di quella riga con l'intero contenuto del file nome-file. Il file in questione viene prima cercato nella directory del file sorgente di partenza, poi in una serie di altre ubicazioni standard. In alternativa, una linea di controllo della forma

#include *<nome-file>*

ricerca solo nelle ubicazioni standard e non nella directory del file sorgente.
I **#include** possono venire nidificati.

12.3 Compilazione condizionale

Una linea di controllo del compilatore della forma

#if *espressione-costante*

verifica se la espressione costante (vedi paragrafo 15) ha valore diverso da zero. Una linea di controllo della forma

#ifdef *identificatore*

verifica se l'identificatore è al momento definito nel preprocessore; cioè se è stato il soggetto di una linea di controllo **#define**. Una linea di controllo della forma

#ifndef *identificatore*

verifica se l'identificatore è al momento non definito nel preprocessore.
Tutte e tre le forme sono seguite da un numero arbitrario di linee, che possono contenere la linea di controllo

#else

e poi la linea di controllo

#endif

Se la condizione verificata è vera, le linee comprese tra **#else** e **#endif** vengono ignorate. Se la condizione verificata è falsa allora tutte le linee comprese tra il test ed un **#else** o, in mancanza di quest'ultimo, **#endif**, vengono ignorate.
Queste costruzioni possono essere nidificate.

12.4 Controllo di linea

A vantaggio di altri preprocessori che generano programmi C, una linea della forma

#line *costante identificatore*

fa in modo che il compilatore creda, al fine di identificare segnalazioni di errore, che il numero della linea successiva del sorgente sia dato dalla costante e che il file di input corrente abbia il nome dell'identificatore. Se l'identificatore è assente, il nome del file originale non cambia.

13. Dichiarazioni implicite

In una dichiarazione non è sempre necessario specificare sia la classe di memoria che il tipo degli identificatori. La classe di memoria nelle definizioni esterne, nelle dichiarazioni di parametri formali e nei membri di strutture, viene data dal contesto. In una dichiarazione all'interno di una funzione, se viene data una classe di memoria senza tipo, si presume che l'identificatore sia **int**; se viene dato un tipo senza classe di memoria, si presume che l'identificatore sia **auto**. Le funzioni rappresentano un'eccezione a quest'ultima regola, poiché le funzioni **auto** sono senza senso (il C non ha la capacità di compilare codice nello stack); se il tipo di un identificatore è "funzione ritornante ...", è implicitamente dichiarato **extern**.
In un'espressione, un identificatore seguito da (che non è già stato dichiarato è contestualmente dichiarato "funzione ritornante **int**".

14. Ancora sui tipi

Questo paragrafo riassume le operazioni che si possono eseguire con oggetti di certi tipo.

14.1 Strutture e unioni

Ci sono solamente due cose che si possono fare con una struttura o unione: chiamare uno dei suoi membri (per mezzo dell'operatore **.**) o prenderne l'indirizzo (per mezzo dell'operatore unario**&**). Altre operazioni, come gli assegnamenti nei due sensi o passarla come parametro, provocano un messaggio di errore. Nel futuro, ci si aspetta che queste operazioni siano disponibili, ma non necessariamente altre.
Il paragrafo 7.1 afferma che in un riferimento diretto on indiretto ad una struttura, (con **.** o **->**) il nome sulla destra deve essere un membro della struttura chiamata o puntata dall'espressione a sinistra. Per permettere una eccezione alle regole di notazione, questa restrizione non viene applicata rigidamente dal compilatore. Infatti, prima di **.** è permesso qualsiasi lvalue, e si presume poi che tale lvalue abbia la forma della struttura di cui fa parte il nome a destra. Inoltre, si richiede che l'espressione prima di **->** sia solamente un puntatore o un intero. Se è un puntatore, viene preso per puntare a una struttura di cui fa parte il nome a destra. Se è un intero, viene preso come l'indirizzo assoluto, in valori di memoria della macchina, dell'appropriata struttura.
Queste costruzioni non sono portabili.

14.2 Funzioni

Si possono fare soltanto due cose con una funzione: chiamarla o prenderne l'indirizzo. Se in un'espressione il nome di una funzione non compare nella posizione di nome-funzione di una chiamata, viene generato un puntatore alla funzione. Quindi, per passare una funzione a un'altra, si può procedere così:

```
int f();
...
g(f);
```

Poi la definizione di **g** può essere

```
g(funcp)
int (*funcp)();
{
    ...
    (*funcp)();
    ...
}
```

Si noti che **f** dev'essere dichiarata esplicitamente nella routine che la chiama poiché la sua comparsa in **g(f)** non era seguita da (.

14.3 Array, puntatori ed indicizzazione

Ogni volta che in un'espressione appare un identificatore di tipo array, viene convertito in un puntatore al primo membro dell'array. A causa di questa conversione, gli array non sono lvalues. Per definizione, l'operatore di indicizzazione **[ ]** viene interpretato in modo che **E1[E2]** è identico a ***((E1)+(E2))**. Per le regole di conversione che si applicano a +, se **E1** è un array e **E2** un intero, allora **E1[E2]** si riferisce all'**E2**-esimo membro di **E1**. Inoltre, nonostante la sua apparenza asimmetrica, l'indicizzazione è un'operazione commutativa.
Nel caso di array a più dimensioni si segue una regola fissa. Se **E** è un array n-dimensionale di ampiezza $i \times j \times ... \times k$, allora una **E** che compare in un'espressione viene convertita in un puntatore a un array a $(n-1)$-dimensioni di ampiezza $j \times ... \times k$. Se a questo puntatore viene applicato esplicitamente o implicitamente l'operatore $*$, come il risultato di un'indicizzazione, il risultato è l'array a $(n-1)$-dimensioni puntato, che viene convertito immediatamente in un puntatore.
Per esempio, consideriamo

```
int x[3] [5];
```

Qui **x** è un array di **3×5** interi. Quando **x** appare in un'espressione, viene convertito in un puntatore all'array (il primo dei tre) di interi di 5 membri. Nell'espressione **x[i]**, che è equivalente a ***(x+i)**, **x** viene prima convertito in un puntatore come descritto; poi **i** viene convertito al tipo di **x**, che implica la moltiplicazione di **i** per la lunghezza dell'oggetto a cui punta il puntatore, cioè 5 oggetti di interi. I risultati sono sommati e si applica un'indirezione per produrre un array (di 5 interi) che di volta in volta viene convertito in un puntatore al primo degli interi. Se c'è un altro indice si applica ancora il medesimo argomento; questa volta il risultato è un intero.
Ne deriva che in C gli array vengono allocati per righe (l'ultimo indice varia più velocemente) e che il primo indice nella dichiarazione aiuta a determinare l'ampiezza di memoria occupata da un array ma non ha altre funzioni nel calcolo degli indici.

14.4 Conversione esplicita di puntatori

Sono permesse alcune conversioni che riguardano i puntatori ma hanno aspetti dipendenti dall'implementazione. Sono tutte specificate per mezzo di un operatore di conversione esplicita di tipo, paragrafi 7.2 e 8.7.
Un puntatore può essere convertito in ognuno dei tipi integral capaci da contenerlo. Il fatto che sia necessario un **int** o un **long** dipende dalla macchina. Anche la funzione di conversione dipende dalla macchina, ma è certo che non ci saranno sorprese per

chi conosce la struttura di indirizzamento della macchina. Più avanti vengono dati i dettagli per alcune macchine particolari.
Un oggetto di tipo integral può essere esplicitamente convertito in un puntatore. La conversione supporta sempre un intero convertito da un puntatore allo stesso puntatore, ma negli altri casi dipende dalla macchina.
Un puntatore ad un certo tipo può essere convertito in un puntatore a un altro tipo. Il puntatore risultante può provocare irregolarità di indirizzamento se il puntatore in questione non si riferisce ad un oggetto opportunamente allineato in memoria. È certo che un puntatore ad un oggetto di una certa dimensione può essere convertito in un puntatore ad un oggetto di dimensione minore e viceversa senza cambiamenti.
Per esempio, una routine di allocazione di memoria può accettare una dimensione (in byte) di un oggetto da allocare, e ritornare un puntatore a **char**; potrebbe essere usato in questo modo.

```
extern char *alloc();
double *dp

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

alloc deve assicurarsi (in un modo che dipende dalla macchina) che il suo valore di ritorno sia adatto a convertire un puntatore in **double**; allora l'*uso* della funzione è portabile.
La rappresentazione dei puntatori sul PDP-11 corrisponde ad un intero di 16 bit ed è misurata in bytes. I **char** non richiedono allineamento; ogni altro oggetto deve avere indirizzo dispari.
Su Honeywell 6000, un puntatore corrisponde ad un intero di 36 bit; la parte per parola è nei 18 bit a sinistra e immediatamente alla loro destra i due bit che selezionano il carattere nella parola. Quindi i puntatori **char** sono misurati in termini di 2^{16} byte; ogni altro oggetto è misurato in termini di 2^{18} parole macchina. Quantità **double** e aggregate che le contengono devono trovarsi in una parola di indirizzo dispari (0 modulo 2^{19}).
L'IBM 370 e l'Interdata 8/32 sono simili. In entrambi, gli indirizzi vengono misurati in byte; gli oggetti elementari devono essere allineati all'interno di un limite uguale alla loro lunghezza, per cui puntatori a **short** devono essere 0 modulo 2, ad **int** e **float** 0 modulo 4 e a **double** 0 modulo 8. Le aggregate sono allineate sui limiti più restrittivi richiesti dagli oggetti che le costituiscono.

15. Espressioni costanti

In parecchi contesti il C richiede espressioni che hanno valore costante: dopo un **case**, come indici di array e nelle inizializzazioni. Nei primi due casi, l'espressione può riguardare solo costanti intere, costanti carattere ed espressioni **sizeof**, possibilmente connesse dagli operatori binari

```
+ - * / % & | ^ << >> == != < > <= >=
```

o dagli operatori unari

```
-   ~
```

o dall'operatore ternario

```
?:
```

Le parentesi possono essere usate per raggruppare, ma non chiamare funzioni.
Per le inizializzazioni è permessa maggior libertà; accanto alle espressioni costanti trattate sopra, si può anche applicare l'operatore unario **&** ad oggetti esterni o statici, e ad array esterni o statici indicizzati con un'espressione costante. L'unario **&** può anche essere applicato implicitamente dalla comparsa di array senza indice e funzioni. La regola basilare è che le inzializzazioni devono avere il valore di una costante o dell'indirizzo di un oggetto precedentemente dichiarato esterno o statico più o meno una costante.

16. Considerazioni di portabilità

Certe parti del C sono dipendenti dalla macchina. Il seguente elenco di possibili problemi non ha certo la pretesa di menzionarli tutti, ma di focalizzarne i principali.
Fattori tipicamente hardware come ampiezza di parole e le caratteristiche dell'aritmetica in floating point e la divisione intera non si sono rivelati nella pratica come un grosso problema. Altri aspetti dell'harware sono riflessi in differenti implementazioni. Alcune di essi, in particolare l'estensione di segno (che converte un carattere negativo in un intero negativo) e l'ordine in cui i byte vengono collocati in una parola, rappresentano un fattore di disturbo che va preso con cautela. La maggior parte degli altri problemi è di minore entità.
Il numero di variabili **register** che può realmente essere collocato nei registri varia da macchina a macchina, come l'insieme dei tipi validi. Comunque, il compilatore si comporta correttamente sulla propria macchina; le dichiarazioni **register** in eccesso o scorrette non vengono considerate.
Insorgono delle difficoltà solamente quando vengono usate dubbie tecniche di programmazione. È imprudente scrivere programmi che dipendono da qualsiasi di queste caratteristiche.
Il linguaggio non specifica l'ordine di valutazione degli argomenti di funzione. È da destra a sinistra sul PDP/11, da sinistra a destra sugli altri sistemi. Non è neanche specificato l'ordine in cui si producono gli effetti collaterali.
Poiché le costanti carattere sono propriamente oggetti di tipo **int**, sono permesse costanti carattere di più caratteri. L'implementazione specifica è strettamente dipendente dalla macchina, poiché l'ordine in cui i caratteri vengono assegnati ad una parola varia da una macchina all'altra.
I campi vengono assegnati alle parole ed i caratteri agli interi da destra a sinistra sul PDP/11 e da sinistra a destra sulle altre macchine. Queste differenze sono invisibili per programmi isolati che non si soffermano in conflitti di tipo (per esempio, convertendo un puntatore a **int** in un puntatore a **char** e ispezionando la memoria puntata), ma dev'essere preso in considerazione quando ci si adegua a situazioni di memoria imposte dall'esterno.
Il linguaggio accettato dai vari compilatori si differenzia per piccoli dettagli. Il maggiore è che il compilatore corrente per il PDP/11 non inizializza strutture che contengono campi di bit, e non accetta un piccolo insieme di operatori di assegnamento in contesti in cui viene usato il valore dell'assegnamento.

17. Anacronismi

Poiché il C è un linguaggio in evoluzione, è possibile trovare alcune costruzioni obsolete nei vecchi programmi. Benché molte versioni del compilatore supportino tali anacronismi, alla fine spariranno lasciando dietro a sé solo un problema di portabilità.
Precedenti versioni del C usavano la forma *=op* invece di *op=* per gli operatori di assegnamento. Ciò porta ad ambiguità; scrivendo

```
x=-1
```

che in effetti decrementa **x** poiché = e – sono adiacenti, ma che può essere facilmente confuso con l'assegnare –1 a **x**.
La sintassi delle inizializzazioni è cambiata: prima, il segno di uguale che introduce un inizializzatore non era presente, così invece di

```
int     = 1
```

si usava

```
int x   1;
```

Il cambiamento è stato apportato poiché l'inizializzazione

```
int   f   (1+2)
```

assomiglia ad una dichiarazione di funzione tanto da confondere i compilatori.

18. Sommario Sintattico

Questo sommario della sintassi del C è volto a facilitarne la comprensione piuttosto che a fornire un'esatta documentazione sul linguaggio

18.1 Espressioni

Le espressioni basilari sono:

espressione:
- *primaria*
- ***** *espressione*
- **&** *espressione*
- **!** *espressione*
- ~*espressione*
- ++ *lvalue*
- – – *lvalue*
- *lvalue* ++
- *lvalue* – –
- **sizeof** *espressione*
- *(nome-di-tipo) espressione*
- *espressione oper-binario espressione*
- *espressione ? espressione* **:** *espressione*
- *lvalue oper-assegn espressione*
- *espressione, espressione*

primaria:
identificatore
costante
stringa
(*espressione* **)**
primaria **(** *elenco-di-espressioni*$_{opz}$ **)**
primaria **[** *espressione* **]**
lvalue **.** *identificatore*
primaria **−>** *identificatore*

lvalue:
identificatore
primaria **[** *espressione* **]**
lvalue **.** *identificatore*
primaria **−>** *identificatore*
***** *espressione*
(*lvalue* **)**

Gli operatori di espressioni primaria

() [] . ->

hanno priorità assoluta e si raggruppano da sinistra a destra. Gli operatori unari

*** & − ! ~ ++ −− sizeof (** *nome-di-tipo* **)**

hanno minore priorità degli operatori primari ma maggiore di ogni operatore binario e si raggruppano da destra a sinistra. Tutti gli operatori binari e gli operatori di condizione si raggruppano da sinistra a destra e hanno priorità decrescente come indicato:

oper-binario:
***** **/** **%**
+ **−**
>> **<<**
< **>** **<=** **>=**
== **!=**
&
^
|
&&
||
?:

Gli operatori di assegnamento hanno tutti la stessa priorità e si raggruppano da destra a sinistra.

oper-assegn:
= **+=** **−=** ***=** **/=** **%=** **>>=** **<<=** **&=** **^=** **|=**

L'operatore virgola ha la priorità minore e si raggruppa da sinistra a destra.

18.2 Dichiarazioni

dichiarazione:
specifica-dich elenco-di-dichiaratori$_{opz}$*;*

specifica-dich:
specifica-di-tipo specifica-dich$_{opz}$
specifica-clas-mem specifica-dich$_{opz}$

specifica-clas-mem:
auto
static
extern
register
typedef

specifica-di-tipo:
char
short
int
long
unsigned
float
double
specifica-di-struttura-o-unione
nome-di-typedef

elenco-dichiaratori-iniz:
dichiaratore-iniz
dichiaratore-iniz, elenco-dichiaratori-iniz

dichiaratore-iniz:
dichiaratore inizializzatore$_{opz}$

dichiaratore:
identificatore
(dichiaratore)
** dichiaratore*
dichiaratore ()
dichiaratore [espressione-costante$_{opz}$ *]*

specifica-strut-o-unione:
struct { *elenco-dich-strut* }
struct *identificatore* { *elenco-dich-strut* }
struct *identificatore*
union { *elenco-dich-strut* }
union *identificatore* { *elenco-dich-strut* }
union *identificatore*

elenco-dich-strut:
dichiarazione-strut
dichiarazione-strut elenco-dich-strut

dichiarazione-strut:
specifica-tipo elenco-dichiaratore-strut;

elenco-dichiaratore-strut:
dichiarazione-strut
dichiaratore-strut, elenco-dichiaratore-strut

dichiaratore-strut:
dichiaratore
dichiaratore : espressione-costante
: espressione-costante

inizializzatore:
= espressione
= { elenco-inizializzatori }
= { elenco-inizializzatori, }

elenco-inizializzatori:
espressione
elenco-inizializzatori, elenco-inizializzatori
{ elenco-inizializzatori }

nome-di-tipo:
specificatore-di-tipo dichiaratore-astratto

dichiaratore-astratto:
vuoto
(dichiaratore-astratto)
** dichiaratore-astratto*
dichiaratore-astratto ()
dichiaratore-astratto [espressione-costante$_{opz}$ *]*

nome-typedef:
identificatore

18.3 Istruzioni

istruzione-composta:
{ elenco-dichiarazioni$_{opz}$ *elenco-istruzioni*$_{opz}$ *}*

elenco-dichiarazioni:
dichiarazione
dichiarazione elenco-dichiarazioni

elenco-istruzioni:
    istruzione
    istruzione elenco-istruzioni

istruzione:
    istruzione-composta
    espressione ;
    if *(espressione) istruzione*
    if *(espressione) istruzione else istruzione*
    while *(espressione) istruzione*
    do *istruzione* **while** *(espressione) ;*
    for *(espressione-*1_{opz}*; espressione-*2_{opz}*; espressione-*3_{opz}*) istruzione*
    switch *(espressione) istruzione*
    case *espressione-costante:*
    default : *istruzione*
    break ;
    continue ;
    return ;
    return *espressione ;*
    goto *identificatore;*
    identificatore:
    ;

18.4 Definizioni esterne

programma:
    definizione-esterna
    definizione-esterna programma

definizione-esterna:
    definizione-di-funzione
    definizione-di-dati

definizione-di-funzione:
    specifica-dich$_{opz}$ *dichiaratore-funzione corpo-funzione*

dichiaratore-di-funzione
    dichiaratore (elenco-di-parametri$_{opz}$ *)*

elenco-di-parametri
    identificatore
    identificatore , elenco-di-parametri

corpo-di-funzione:
    elenco-dichiarazioni istruzione-composta

istruzione-di-funzione:
    { *elenco-dichiarazioni*$_{opz}$ *elenco-istruzioni* }

18.5 Preprocessore

#**define** *identificatore stringa-di-token*
#**define** *identificatore (identificatore,...,identificatore)*
stringa-di-token
#**undef** *identificatore*
#**include** *“nome-file”*
#**include** *‹nome-file›*
#**if** *espressione-costante*
#**ifdef** *identificatore*
#**ifndef** *identificatore*
#**else**
#**endif**
#**line** *costante identificatore*

INDICE ANALITICO

\ carattere backslash 7,33,166
\ carattere di escape 7,33,166
& operatore **AND** su bit 42,176
+ operatore di addizione 35,174
~ operatore di complamento a uno 43, 173
/ operatore di divisione 10,35,174
* operatore di indirezione 83,173
& operatore di indirizzo 83,173
- operatore di meno unario 35,174
* operatore di moltiplicazione 35,174
! operatore di negazione logica 36,173
- operatore di sottrazione 35,174
> operatore maggiore di 36,175
< operatore minore di 36,175
% operatore modulo 35,174
^ operatore **OR** esclusivo su bit 42,176
| operatore **OR** inclusivo su bit 42,176
?: espressione condizionale 45,177
&& operatore di **AND** logico 19,36,43,177
-- operatore di decremento 16,40,95,173
!= operatore di diseguaglianza 14,36, 175
++ operatore di incremento 16,40,95,173
|| operatore di **OR** logico 19,36,43,177
-> operatore di puntatore a struttura 114, 171
>> operatore di shift a destra 43,173
<< operatore di shift a sinistra 43,175
== operatore di uguaglianza 17,36,176
>= operatore maggiore o uguale a 36,175
<= operatore minore o uguale a 36,175
\Ø carattere nullo 26,33

accesso, modalita' su file 141,150
addittivi, operatori 174
addizione, operatore, + 35,174
aggiungi, funzione 126
aggregate 184
albero binario 121
albero, funzione 123
alfabetico, ordinamento 107
allineamento attraverso **union** 160
allineamento, campo 128,182
allineamento, restrizioni 121,123,130,
145,159,197
alloc, funzione 91,161
allocatore, memoria 90,123,126,197
ambiguita', **if-else** 50,188
ampiezze dei numeri 9,16,32,168
anacronismo 200
anno bisestile, calcolo 35
a.out 6,64
apice singolo, carattere 17
argc conteggio di argomenti 102
argomenti, **#define** 81
argomenti, linea di comando 102
argomenti, numero variabile di 66
argomento, conversione 40
argomento, funzione 6,22,172
argomento, lista 6
argomento, puntatore 89
argomento, sottoarray 89
argv vettore di argomenti 102
aritmetica, conversioni 37,170
aritmetica, operatori 35,174
aritmetica, puntatori 84,88,90,106,120,
174
aritmetica, tipi 169
array, a due dimensioni 96
array, a piu' dimensioni 96,197
array, ampiezza di default 78,101,116,
181
array, caratteri 18,24,93
array, conversione del nome 88,171
array di puntatori 99

array di strutture 115
array, dichiarazione 20,181
array e puntatori 87,102
array, indici a due dimensioni 97
array, inizializzazione 78,101,184
array, nome degli argomenti 3,23,66,89,
99,192
array, ordine in memoria 97,197
array, spiegazione degli indici 87,197
ASCII set di caratteri 17,33,38,118
asm, parola chiave 166
assegnamenti multipli 19
assegnamento, conversione 39,177
assegnamento, espressione 15,19,45,177
assegnamento, istruzione nidificata 15,
19
assegnamento, operatori 36,44,177
assegnamento, soppressione, scanf 138
associativita' degli operatori 46,171
atof funzione 65
atoi funzione 37,55
auto classe di memoria 22,27,178
automatiche, inizializzazione 27,35,76,
184
automatiche, variabili 22,27,67
automatiche, visibilita' 76
\b carattere di backspace 7
backslash, carattere \ 7,33,166
bidimensionale, array 97
bidimensionale, indici di array 98
bidimensionale, inizializzazione,array 99
binaria, funzione di ricerca 52,117,120
binario, albero 121
bit, campi 126,182
bit, metodi di manipolazione 127
bit operatore **AND**,.& 42,175
bit operatore **OR** esclusivo, ^ 42,175
bit operatore **OR** inclusivo, ¦ 42,175
bit operatori 42,175
bitcount funzione 44
blocchi, inizializzazioni in 188
blocco 49,75,188

blocco, struttura 3,49,75,188
break, istruzione 53,57,190
bufferizzato **getchar** 150
bufferizzato input 148
byte 118,174
%c conversione 11
C preprocessore 80
C programma di verifica (**lint**) 3
calcolatore programma 66,67,70,139
calloc allocatore di memoria 145
campi, restrizioni 128,183
campo, allineamento 128,183
campo, dichiarazione 128,183
campo, dimensione 127
carattere, \ escape 33
carattere, array 18,24,93
carattere con segno 38,168
carattere, con segno 38,168
carattere costante 17,33,166
carattere costante, ottale 33
carattere, funzione di test 145
carattere, inizializzazione di array 78, 186
carattere, input/output 13
carattere, negativo 169
carattere, programma di conteggio 16
carattere, puntatore 93
carattere, set **ASCII** 17,33,38,118
carattere, set **EBCDIC** 33,38
carattere, stringa 7,34,74,167
carattere-intero, conversione 21,37,169
caricare dalla libreria standard 134
caricare piu' files 64
case prefisso 52,189
cast operatore 40,124,146,173,186
cat programma 140,142,143
cc comando 6,64,133
cerca, funzione 125
cfree 146
char, tipo 9,32,167,179
chiamata per indirizzo 3,23,85
chiamata per valore 3,23,66,85,172

ciclo, **for** 11
ciclo, infinito 54
ciclo, test alla fine 56
ciclo, test all'inizio 17,56,101
ciclo, **while** 9
close chiamata di sistema 150
coercizione, operatore 40
commento 8,165
commutativi, operatori 35,47,171
compilare, files multipli 64
compilare, programma **C** 6,23
compilatore, linee di controllo 194
compilazione separata 3,61,71,192
composta, istruzione 49,75,188
condizionale, compilazione 195
condizionale, espressione, ?: 45,177
confronto, puntatori 92,175
confronto, separati 3,61,71,192
conteggio di parole, programma 18,121
continue, istruzione 58,190
continue, istruzione 58,190
contorno, condizioni di 17,57,101
conversione, argomento di funzione 40, 172
conversione attraverso **return** 66,190
conversione attraverso un assegnamento 39, 177
conversione, carattere-intero 21,37,169
conversione di funzione 171
conversione di nome di array 88,171
conversione, **double-float** 40,170
conversione, **float-double** 39,65,170
conversione, floating-intero 40,170
conversione, formato 10
conversione, input 86
conversione, intero-carattere 39,169
conversione, intero-floating 10,170
conversione, intero-**long** 169
conversione, intero-puntatore 174,197
conversione, intero-**unsigned** 170
conversione, intero-unsigned 170
conversione, **long-intero** 169

conversione, operatore, esplicito 173,
197
conversione, puntatore 123,174,197
conversione, puntatore-intero 96,170,174,
197
conversione, regole 39
conversione, valore di verita' 39
conversioni, aritmetica 37,170
copy, funzione 25
copy, programma 15
costante 166
costante, carattere 17,33,166
costante, **double** 167
costante, esadecimale 33,166
costante, espressione 33,52,198
costante, floating 10,167
costante, intera 10,166
costante, **long** 33,166
costante, ottale 33,166
costante, simbolica 12,14,18,80,118
cp, programma 151
creat, chiamata di sistema 150
%d, conversione 11
data, conversione 97,111,113
dati, definizione 192
dati, struttura ricorsiva 122
decisioni a piu' vie 21,51
decremento, operatore, -- 16,40,96,173
default, ampiezza di un array 78,101,116,
181
default, inizializzazione 78,184
default, prefisso 52,190
#define 12,80,194
#define argomenti 81
#define, troppo lungo 80
definizione di variabile esterna 29
definizione, esterna 27,191
definizione, funzione 22,63,191
derivati, tipi 3,9,169
descrittore, file 147
dichiaratore 180
dichiaratore, astratto 185

dichiarazione, array 20,181
dichiarazione, campo 127,182
dichiarazione, classe di memoria 178
dichiarazione di funzione 180
dichiarazione di funzione, implicita 64,
172,196
dichiarazione di puntatore 84,89,180
dichiarazione di variabile esterna 27
dichiarazione e definizione 71
dichiarazione, **extern** 29
dichiarazione, **FILE** 140
dichiarazione, implicita 195
dichiarazione, obbligatoria 8,34,71
dichiarazione, **register** 75
dichiarazione, sintassi 34
dichiarazione, **static** 74
dichiarazione, struttura 111,182
dichiarazione, tipo 180
dichiarazione, **typedef** 130,179,187
dichiarazione, **union** 128,182
dichiarazioni 178
difesa di programmazione 51,53
dipendenza, macchina 38,128,129,131
directory 156
dir.h struttura di directory 156
disuguaglianza, operatore != 14,36,175
divisione, intera 10,35,200
divisione, operatore, / 10,35,174
DO, idioma del ciclo 54
do, istruzione 3,56,189
doppi apici, carattere 7,18,34
double costante 167
double, tipo 9,16,32,179
double-float, conversione 40,170
due dimensioni, array 97
due dimensioni, indici di array 98
due dimensioni, inizializzazione,array 99
E, notazione 32
EBCDIC, set di caratteri 33,38
echo, programma 103
effetti collaterali 48,81,171
efficienza 45,61,75,79,101,118,123,133,148,162

eguaglianza, operatore == 17,36,175
eguaglianza, operatori 36,175
#else 195
else, istruzione 19
else-if 21,51
end of file 14,134
#endif 195
entry 166
EOF, valore di 14,38,134
esadecimale, costante 33,166
escape, carattere \ 7,33,166
escape, sequenza 7,17,34,166
esplicito, conversione di tipo 40,123, 145
esplicito, operatore di conversione 173, 197
espressione 173
espressione, assegnamento 15,19,45,177
espressione, costante 33,52,198
espressione, indici 20
espressione, istruzione 49,51,188
espressione, primaria 171
espressione, tipo della 12
espressione, tra parentesi 172
espressione, unaria 173
espressioni, ordine di valutazione 47, 171
estensione del segno 38,39,153,200
esterna, definizione 27,191
esterna, definizione della variabile 29
esterna, dichiarazione della variabile 27
esterna, variabile 27,67
esterne, inizializzazioni 35,67,72,76
esterne, visibilita' 71,193
esterno, definizione di dati 192
esterno, lunghezza dei nomi 31,165
esterno, variabili **static** 74
etichetta 58,191
etichettata, istruzione 191
exit, _exit 143
extern, classe di memoria 178

extern, dichiarazione 29
%f, conversione 11,16
fa-niente, funzione 63
fclose 142
fgets 144
file, accesso 140,147
file, apertura 140,147
file, appendere 141,152
file, concatenazione 140
file, creazione 141,147
file, descrittore 147
FILE, dichiarazione 140
file, end of 14,134
file, inclusione 80,194
file, modalita' di accesso 141,150
file, modalita' di protezione 150
file, programma copy 14,149
file, puntatore 140,153
file, ritorno all'inizio 152
_fillbuf, funzione 155
fine del file 14,134
fine della stringa 26,33
float, tipo 9,32,179
float-double, conversione 39,65,170
fondamentali, tipi 3,9,173
fopen 140
fopen, funzione 154
for, istruzione 3,11,16,53,189
formale, parametro 22,75,89,191
formato, conversione 10
formato libero di input 86
formattato, input 11,137
formattato, output 10,135
fortran, parola chiave 166
fprintf 141
fputs 144
free, funzione 91,163
fscanf 141
fsize, programma 157
funzione, argomenti 6,22,172
funzione, conversione 171
funzione, conversione degli argomenti 40,172

funzione, definizione 22,63,191
funzione, dichiarazione 180
funzione, dichiarazione implicita 64,172,
196
funzione, nome, lunghezza 31,165
funzione, puntatore a 107,131,196
funzione, semantica di chiamata 172
funzione, sintassi di chiamata 172
funzione, valore di ritorno 22
funzioni di testing sui caratteri 145
funzioni, operazioni permesse sulle 196
getbits, funzione 43
getc 141
getch, funzione 74
getchar 13,134
getchar, bufferizzato 150
getchar, macro 141
getchar, senza buffer 149
getchar, valore di 38
getint, funzione 87
getline, funzione 25,63
getop, funzione 118
getword, funzione 118
giorno_anno, funzione 97,114
goto, istruzione 58,191
graffe 6,9,49
graffe, posizioni delle 10
grep, programma 61,104
hash, funzione 125
hash, tavola 124
#if 195
if, istruzione 17
#ifdef 195
if-else, ambiguita' 50,95
if-else, istruzione 3,19,49,188
#ifndef 195
illecita, aritmetica dei puntatori 92,
93
implicita, dichiarazione 195
implicita, dichiarazione di funzione 64,
172,196

#include 80,133,194
inconsistente, dichiarazione di tipo 66
incremento, operatore ++ 16,40,96,173
indentazione 10,16,50
index, funzione 63
indice, espressione 20
indice, inizio 20
indici e puntatori 83
indicizzare, spiegazione 87,197
indirezione, operatore * 83,173
indirizzi, aritmetica 3,90
indirizzo di una variabile 23,83
indirizzo, operatore & 83,173
infinito, ciclo 54
infissa, notazione 67
inizializzatore, forma permessa 198
inizializzazione 35,184
inizializzazione, array 77,101,184
inizializzazione, array bidimensionali 99
inizializzazione, array di strutture 116
inizializzazione, automatiche 27,35,76,
184
inizializzazione, default 78,184
inizializzazione, esterne 35,67,72,76
inizializzazione in blocchi 188
inizializzazione, puntatore 91,120
inizializzazione, registri 76
inizializzazione, statiche 35,76,184
inizializzazione, struttura 112,113,184
inode 156
input, bufferizzato 148
input, carattere 13
input, conversione 86
input e output, terminale 13,134,148
input, formattato 11,137
input, redirezione 134
input, rimandarlo indietro 74
input, senza buffer 148
int tipo 9,32,179
integral, tipo 169
intera, costante 10,166
Interdata **UNIX** 2

interni, lunghezza dei nomi 31
interni, variabili **static** 74
intero-carattere, conversione 39,169
intero-floating, conversione 10,170
intero-long, conversione 169
intero-puntatore, conversione 174,197
intero-unsigned, conversione 170
inversa, notazione Polacca 67
I/O, redirezione 134,141,148
isalpha macro 118,145
isdigit macro 118,145
islower macro 145
isspace macro 145
istruzioni 188
istruzioni, raggruppate 3,49
istruzioni, sequenza di 188
isupper macro 135,145
itoa, funzione 56
%ld conversione 16
leggibilita', programma 10,15,19,31,37,
45,50,54,57,77,79,131
lessicale, visibilita' 192
lessicali, convenzioni 165
libreria, funzione 6,7,61,71
libreria, standard 4,86,90,118,133,147,
153
#line 195
linea di comando, argomenti 103
linee, programma di conteggio 17
lint, programma di verifica 3,48,64,97,
131
lista della directory, programma 156
lista di parole chiave 166
locale, variabile 27
logica, valore dell'espressione 39
logico, operatore 36
logico, operatore **AND** && 19,36,43,177
logico, operatore di negazione ! 36,173
logico, operatore OR 19,36,43,177
long, costante 33,166
long, tipo 9,16,32,168,179
long-intero, conversione 169

loop, for 11
loop, infinito 54
loop, test alla fine 56
loop, test all'inizio 17,56,101
loop, while 9
lower, funzione 38
ls, comando 156
lseek, chiamata di sistema 152
lunghezza dei nomi 31,165
lunghezza della stringa 26,34,93
lvalore 169
macchina, dipendenza 36,38,43,75,92,97,
121,123,128,129,131,153,159,200
macro con argomenti 81
macro, preprocessore 80,194
maggiore di, operatore > 36,175
maggiore o uguale a, operatore >= 36,175
magico, numero 12
main, funzione 6
mancante, specifica classe di memoria 179
mancante, specifica del tipo 179
membro, nome, struttura 112,183
memoria, allocatore 90,123,126,197
memoria, allocatore, calloc 146
memoria, allocazione degli array 98,197
memoria, classe **auto** 22,27,179
memoria, classe **extern** 178
memoria, classe **register** 75,178
memoria, classe **static** 27,74,179
memoria, dichiarazione di classe 179
memoria, specifica di classe 179
memoria, specifica di classe, mancante 179
mese_giorno, funzione 97,115
minore di, operatore < 36,175
minuscole, programma 135
modello, struttura 111,183
modello, unione 183
modularizzazione 22,24,29,67,68,99
modulo, operatore % 35,174
moltiplicativi, operatori 174
moltiplicazione, operatore * 35,174
morecore, funzione 162

multi-dimensionali, array 97,127
multipli, assegnamenti 19
multipli files, compilare 64
mumeri, grandezza dei 9,16,32,168
\n carattere di newline 7,17
negativo, carattere 38,169
newline, carattere \n 7,17
nidificata, istruzione di assegnamento 15,
    19
nidificata, struttura 112
nome, simbolico 12
nome_mese, funzione 102
nomi, lunghezza 31,165
NULL, puntatore 91,178
nulla, istruzione 16,191
nulla, stringa 34
nullo, carattere \0 26,33
numcmp, funzione 110
numerico, ordinamento 107
numero di argomenti, variabile 66
%o conversione 11
open, chiamata di sistema 150
operatori, additivi 174
operatori, aritmetici 35,174
operatori, assegnamento 36,44,177
operatori, associativita' 46,171
operatori, commutativi 35,47,171
operatori, logici 36
operatori, moltiplicativi 174
operatori, priorita' degli 15,46,85,114,
    115,171,201
operatori, relazionali 14,36,175
operatori, shift 42,175
operatori, sui bit 42,175
operatori, uguaglianza 36,175
operazioni permesse sui puntatori 93
operazioni permesse sulle funzioni 196
operazioni permesse sulle strutture 196
operazioni permesse sulle unioni 196
ordinamento, alfabetico 107
ordinamento, linee di testo 99
ordinamento, numerico 107

ordinamento, programma 99,108
ordine di valutazione 19,35,43,47,48,56,
70,81,85,171,200
ordine di valutazione, espressioni 47,
171
origine, indice 20
ottale, costante 33,166
ottale, costante carattere 33
output, carattere 13
output, formattato 10,135
output, redirezione 134
overflow 36,171
parametri, formali 22,75,89,191
parentesi, espressione 172
parole chiave, conteggio delle 117
parole chiave, elenco 166
parole riservate 31,166
pattern, programma di ricerca 62,104,105
PDP-11 UNIX 2
permesse, forme di inizializzatori 198
permessi sulle funzioni, operazioni 196
permessi sulle strutture, operazioni 196
permessi sulle unioni, operazioni 196
pipe 134,148
Polacca, notazione 67
pop funzione 70
portabilita' 2,3,33,38,43,66,97,129,131,
133,135,153,159,200
posizione delle graffe 10
postfissi ++ e -- 40,95,173
potenza 22
potenza, funzione 22,23
prefissi ++ e -- 40,96,173
preprocessore **C** 80
preprocessore, macro 80,194
primaria, espressione 171
printd, funzione 79
printf 7,11,43
priorita' degli operatori 15,46,85,114,
115,171,201
programma, argomenti 102
programma, blank in coda 57

programma, calcolatore 66,67,70,139
programma, **cat** 140,142,143
programma, conteggio caratteri 16
programma, conteggio linee 17
programma, conteggio parole 18,121
programma, conteggio parole chiave 117
programma, conteggio spazi bianchi 20, 52
programma, conversione di temperature 8
programma, copia 15
programma, copia file 14,149
programma, **cp** 151
programma di verifica, **lint** 3,48,64,97, 131
programma, **echo** 103
programma, formato 10,16,121,165
programma, **fsize** 157
programma, **grep** 61,104
programma, leggibilita' 19,37,54,57,79
programma, lettere minuscole 135
programma, linee piu' lunghe 25
programma, lista directory 156
programma, ordinamento 99,108
programma, ricerca in tavola 124
programma, ricerca pattern 62,104,106
protezioni, modalita' su file 150
puntatore a funzione 107,131,196
puntatore a struttura 119
puntatore, argomento 89
puntatore, aritmetica 84,88,90,106,120, 179
puntatore, aritmetica, non-permessa 92, 93
puntatore, aritmetica, scalare 88,92
puntatore, carattere 93
puntatore, confronto 92,175
puntatore, conversione 123,174,197
puntatore, dichiarazione 84,89,180
puntatore, file 140,153
puntatore, inizializzazione 91,120
puntatore, **NULL** 91,178
puntatore, sottrazione 92

puntatore-intero, conversione 96,170,174,
197
puntatori 3,23
puntatori, array di 99
puntatori ed array 87,102
puntatori ed indici 87
puntatorj, operazioni permesse 93
punto e virgola 9,13,16,49,51
push, funzione 70
putc 141
putc, macro 153
putchar 13,134
putchar, macro 141
read, chiamata di sistema 148
readlines, funzione 100
redirezione di I/O 134
register, classe di memoria 75,178
register, dichiarazione 75
registri, inizializzazione 76
regole, conversione di tipo 39,170
regole di visibilita' 71,192
relazionali, espressioni, valore delle 39
relazionali, operatori 14,36,175
restrizioni sui campi 128,183
restrizioni sui registri 200
restrizioni sui tipi 181
restrizioni sulle strutture 113,196
restrizioni sulle unioni 130,196
return, conversione da 66
return, conversione di tipo da 66,190
return, istruzione 22,64,66,190
reverse, funzione 56
riapertura di file 152
ricerca in tavola, programma 124
ricorsione 3,78,122,172
ricorsiva, struttura di dati 122
ricorsive, strutture 122,183
riservate, parole 31,166
ritorno dell'input 74
%s conversione 11,26
sbrk, chiamata di sistema 161
scalare nell'aritmetica dei puntatori 88,92

scanf 137
scanf, soppressione dell'assegnamento 138
scientifica, notazione 32
seek, chiamata di sistema 152
semantica, strutture ricorsive 172
senza buffer, **getchar** 149
senza buffer, input 148
separata, compilazione 3,61,71,192
sequenza di istruzioni 188
Shell, ordinamento 55,101
shift a destra, operatore >> 43,175
shift a sinistra, operatore << 43,175
shift, operatori 42,175
short, tipo 9,32,168,179
simboli, rimpiazzamento 194
simbolica, costante 12,14,18,80,118
simbolica, costante, lunghezza 31
sintassi dei nomi delle variabili 31
sintassi delle dichiarazioni 34
sintassi, notazione 168
sintassi, riferimenti a struttura 172
sintassi, sommario 200
sizeof, operatore 118,173
sort, funzione 55,101,109
sottoarray, argomento 89
sottrazione, operatore - 35,174
sottrazione, puntatore 92
spazi bianchi, programma di conteggio 20, 52
spazi in coda, programma 57
specificatore, classe di memoria 179
specificatore, classe memoria mancante 179
specificatore, tipo 179
specificatore, tipo mancante 179
sprintf 139
squeeze, funzione 41
sscanf 139
stampalbero, funzione 123
standard error 148
standard input 134,148
standard libreria 4,86,90,118,133,147, 153

standard libreria, caricamento da 134
standard output 134,148
stat, chiamata di sistema 157
stat, definizioni 157
stat, struttura 157
stat.h 157
static, classe di memoria 27,74,179
static, dichiarazione 74
static, variabili esterne 74
static, variabili interne 74
statiche, inizializzazione 36,76,184
stderr 141,143
stdin 141
stdio.h, contenuti 153
stdio.h, file di intestazione 133
stdout 141
strcat, funzione 42
strcmp, funzione 95
strcpy, funzione 94
stringa costante 7,18,33,93,167
stringa, fine della 26,33
stringa, lunghezza 26,34,93
stringa, tipo 172
strlen, funzione 34,89,92
strsave, funzione 96
struttura di array, inizializzazione 116
struttura, dichiarazione 111,182
struttura, inizializzazione 112,113,184
struttura, modello 111,183
struttura, nidificata 112
struttura, nome dei membri 112,183
struttura, operatore di membro . 112,171
struttura, operatore puntatore -> 114, 171
struttura, operazioni permesse 196
struttura, puntatore a 119
struttura, restrizioni 113,196
struttura, ricorsiva 122,183
struttura, semantica di riferimenti 172
struttura, sintassi di riferimenti 172
strutture, array di 115

swap, funzione 86,110
switch, istruzione 3,22,52,68,189
system 146
tab, carattere 7
terminale, input e output 13,134,148
terminatore di istruzione 9,49
termine, programma 142,143
test alla fine del ciclo 56
test all'inizio del ciclo 17,56,101
testo, ordinamento di linee 99
tipi, aritmetica 169
tipi derivati 3,9,169
tipi fondamentali 3,9,173
tipi in virgola mobile 169
tipi interi 169
tipo, analisi 3,123
tipo **char** 9,32,168,179
tipo, conflitto 66
tipo, conversione attraverso **return** 66, 190
tipo, conversione esplicita 40,123,146
tipo, conversioni 37
tipo di espressione 12
tipo di stringa 172
tipo di una variabile 8
tipo, dichiarazione 180
tipo, dichiarazione inconsistente 66
tipo, **double** 9,16,32,179
tipo, **float** 9,32,179
tipo, **int** 9,32,179
tipo, **long** 9,16,32,168,179
tipo, nomi 185
tipo, operatore di conversione 40
tipo, regole di conversione 39,170
tipo, restrizioni 181
tipo, **short** 9,32,168,179
tipo, specificazione 179
tipo, specificazione mancante 179
tipo, **unsigned** 32,43,169,179
token, rimpiazzamento 194
tolower, macro 135,145
toupper, macro 145

troncare 10,35,66
troppo lungo, **#define** 80
type, funzione 118
typedef, uso 159
types.h 157
unaria, espressione 173
unario, operatore meno - 35,173
#undef 194
underflow 36
underscore, carattere 31,165
ungetc 145
ungetch, funzione 74
union, allineamento 160
union, dichiarazione 128,182
union, modello 183
unioni, operazioni permesse 196
unioni, restrizioni 130,196
UNIX, file system 147,156
UNIX, Interdata 2
UNIX, PDP/11 2
UNIX, sistema operativo 2,4
unlink, chiamata di sistema 151
uno, operatore di complemento a ~ 43,173
unsigned, tipo 32,43,169,179
unsigned-intero, conversione 170
uso di **typedef** 159
valore di EOF 14,38,134
valore di espressioni logiche 39
valore di espressioni relazionali 39
valore di getchar 38
valore di verita', conversione 39
valutazione, ordine della 19,35,43,47,
48,56,70,81,85,171,200
variabile, automatica 22,27,67
variabile, esterna 22,67
variabile, indirizzo della 23,83
variabile, lunghezza dei nomi 31
variabile, numero di argomenti 66
variabile, sintassi dei nomi 31
virgola mobile, costante 10,167
virgola mobile, tipi 169
virgola mobile-interi, conversione 40,170

virgola, operatore 55,178
visibilita' delle automatiche 71
visibilita' delle esterne 71,193
visibilita', regole 71,192
vuota, funzione 63
vuota, istruzione 16,191
while, istruzione 3,9,53,189
write, chiamata di sistema 148
writelines, funzione 100,101
%x, conversione 11
zero, omissione del test su 50,93,95

Il linguaggio C deve la sua attuale fortuna al fatto che gran parte dei programmi applicativi funzionanti sotto UNIX, ed il sistema operativo UNIX stesso, sono scritti in C: la grande diffusione di UNIX ha quindi contribuito ad aumentare il numero di utenti e tecnici dell'informatica che usano questo linguaggio.
IL C è un linguaggio di applicazione generale, vale a dire che può essere usato per scrivere programmi di elaborazione numerica ma anche software di base, data base, word processing e così via.
Il linguaggio presenta una estrema portabilità: conoscerlo significa quindi essere in grado di realizzare programmi sempre più indipendenti dalla singola macchina e con un alto grado di efficienza esecutiva, pregi non indifferenti nel panorama attuale dell'informatica.
Lo scopo di questo libro è aiutare il lettore a imparare a programmare in C; non è un manuale introduttivo alla programmazione e presuppone una certa familiarità con i concetti basilari della materia, ma poiché il C è facile da imparare e ben si adatta a diversi livelli di esperienze, anche il programmatore principiante dovrebbe essere in grado di procedere nella lettura.

L. 25.000

COD 541 P ISBN 88-7056-211-5

180

LINGUAGGIO C

Brian W. Kernighan
Dennis M. Ritchie

GRUPPO EDITORIALE JACKSON